Máster Universitario en Ingeniería de Telecomunicación

Trabajo fin de Máster

Técnicas Avanzadas de Modelado Computacional en Superconductividad Aplicada

Author
Elena Conderana Medem

Supervised by
Carlos José Hernando López de Toledo

Madrid
Julio 2025

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

TÉCNICAS AVANZADAS DE MODELADO COMPUTACIONAL EN SUPERCONDUCTIVIDAD APLICADA

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2024/25 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Elena Conderana Medem Fecha: 05/ 07/ 2025

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Carlos José Hernando López de Toledo Fecha:06/ 07/ 2025

AUTORIZACIÓN PARA LA DIGITALIZACIÓN, DEPÓSITO Y DIVULGACIÓN EN RED DE PROYECTOS FIN DE GRADO, FIN DE MÁSTER, TESINAS O MEMORIAS DE BACHILLERATO

1º. Declaración de la autoría y acreditación de la misma.

El autor D. Elena Conderana Medem

DECLARA ser el titular de los derechos de propiedad intelectual de la obra:Técnicas avanzadas de modelado computacional en superconductividad aplicada, que ésta es una obra original, y que ostenta la condición de autor en el sentido que otorga la Ley de Propiedad Intelectual.

2º. Objeto y fines de la cesión.

Con el fin de dar la máxima difusión a la obra citada a través del Repositorio institucional de la Universidad, el autor **CEDE** a la Universidad Pontificia Comillas, de forma gratuita y no exclusiva, por el máximo plazo legal y con ámbito universal, los derechos de digitalización, de archivo, de reproducción, de distribución y de comunicación pública, incluido el derecho de puesta a disposición electrónica, tal y como se describen en la Ley de Propiedad Intelectual. El derecho de transformación se cede a los únicos efectos de lo dispuesto en la letra a) del apartado siguiente.

3º. Condiciones de la cesión y acceso

Sin perjuicio de la titularidad de la obra, que sigue correspondiendo a su autor, la cesión de derechos contemplada en esta licencia habilita para:

a) Transformarla con el fin de adaptarla a cualquier tecnología que permita incorporarla a internet y hacerla accesible; incorporar metadatos para realizar el registro de la obra e incorporar "marcas de agua" o cualquier otro sistema de seguridad o de protección.
b) Reproducirla en un soporte digital para su incorporación a una base de datos electrónica, incluyendo el derecho de reproducir y almacenar la obra en servidores, a los efectos de garantizar su seguridad, conservación y preservar el formato.
c) Comunicarla, por defecto, a través de un archivo institucional abierto, accesible de modo libre y gratuito a través de internet.
d) Cualquier otra forma de acceso (restringido, embargado, cerrado) deberá solicitarse expresamente y obedecer a causas justificadas.
e) Asignar por defecto a estos trabajos una licencia Creative Commons.
f) Asignar por defecto a estos trabajos un HANDLE (URL *persistente)*.

4º. Derechos del autor.

El autor, en tanto que titular de una obra tiene derecho a:

a) Que la Universidad identifique claramente su nombre como autor de la misma
b) Comunicar y dar publicidad a la obra en la versión que ceda y en otras posteriores a través de cualquier medio.
c) Solicitar la retirada de la obra del repositorio por causa justificada.
d) Recibir notificación fehaciente de cualquier reclamación que puedan formular terceras personas en relación con la obra y, en particular, de reclamaciones relativas a los derechos de propiedad intelectual sobre ella.

5º. Deberes del autor.

El autor se compromete a:

a) Garantizar que el compromiso que adquiere mediante el presente escrito no infringe ningún derecho de terceros, ya sean de propiedad industrial, intelectual o cualquier otro.
b) Garantizar que el contenido de las obras no atenta contra los derechos al honor, a la intimidad y a la imagen de terceros.
c) Asumir toda reclamación o responsabilidad, incluyendo las indemnizaciones por daños, que pudieran ejercitarse contra la Universidad por terceros que vieran infringidos sus derechos e

intereses a causa de la cesión.

d) Asumir la responsabilidad en el caso de que las instituciones fueran condenadas por infracción de derechos derivada de las obras objeto de la cesión.

6º. Fines y funcionamiento del Repositorio Institucional.

La obra se pondrá a disposición de los usuarios para que hagan de ella un uso justo y respetuoso con los derechos del autor, según lo permitido por la legislación aplicable, y con fines de estudio, investigación, o cualquier otro fin lícito. Con dicha finalidad, la Universidad asume los siguientes deberes y se reserva las siguientes facultades:

- La Universidad informará a los usuarios del archivo sobre los usos permitidos, y no garantiza ni asume responsabilidad alguna por otras formas en que los usuarios hagan un uso posterior de las obras no conforme con la legislación vigente. El uso posterior, más allá de la copia privada, requerirá que se cite la fuente y se reconozca la autoría, que no se obtenga beneficio comercial, y que no se realicen obras derivadas.
- La Universidad no revisará el contenido de las obras, que en todo caso permanecerá bajo la responsabilidad exclusive del autor y no estará obligada a ejercitar acciones legales en nombre del autor en el supuesto de infracciones a derechos de propiedad intelectual derivados del depósito y archivo de las obras. El autor renuncia a cualquier reclamación frente a la Universidad por las formas no ajustadas a la legislación vigente en que los usuarios hagan uso de las obras.
- La Universidad adoptará las medidas necesarias para la preservación de la obra en un futuro.
- La Universidad se reserva la facultad de retirar la obra, previa notificación al autor, en supuestos suficientemente justificados, o en caso de reclamaciones de terceros.

Madrid, a 5 de julio de 2025

ACEPTA

Fdo. 

Motivos para solicitar el acceso restringido, cerrado o embargado del trabajo en el Repositorio Institucional:

Máster Universitario en Ingeniería de Telecomunicación

Trabajo fin de Máster

Técnicas Avanzadas de Modelado Computacional en Superconductividad Aplicada

Author
Elena Conderana Medem

Supervised by
Carlos José Hernando López de Toledo

Madrid
Julio 2025

TÉCNICAS AVANZADAS DE MODELADO COMPUTACIONAL EN SUPERCONDUCTIVIDAD APLICADA

Autor: Conderana Medem, Elena.
Director: Hernando López de Toledo, Carlos José.
Entidad Colaboradora: Cyclomed Technologies

RESUMEN DEL PROYECTO

Este estudio propone un modelo para el cálculo de pérdidas AC en un sistema SMES durante transitorios eléctricos, implementado en Julia y optimizado para GPU. La validación del modelo frente a la implementación base en MATLAB revela una mejora en el cómputo de las matrices de potencial eléctrico y campo magnético superiores al 95%, así como una convergencia mejorada durante la resolución de ecuaciones diferenciales. El modelo final simplifica la definición de la geometría y los transitorios a simular, agilizando exitosamente el proceso de diseño de un sistema SMES.

Palabras clave: SMES, HTS, pérdidas AC, Julia, GPU, simulación numérica

1. Introducción

Los sistemas SMES almacenan energía en forma de campo magnético [1]. Para ello se hace circular corriente por una bobina superconductora hasta alcanzar una determinada intensidad. Alcanzado este punto la bobina enfriada criogénicamente se conecta en cortocircuito, almacenando de manera indefinida la energía generada en forma de campo magnético, pues la resistividad nula del superconductor no genera pérdidas de corriente. Sin embargo, uno de los grandes retos de esta tecnología reside en las pérdidas generadas durante los ciclos de carga y descarga en corriente alterna (AC) que incrementan la carga térmica del sistema [2]. Para garantizar el funcionamiento óptimo, manteniendo la temperatura del superconductor por debajo del umbral crítico, es fundamental simular los transitorios eléctricos para poder medir el impacto y monitorizar su estado en tiempo real, aunque se trata de un proceso complejo con un elevado coste computacional [2].

Ante el desafío que representa el tiempo de cómputo, esta tesis tiene como objetivo reducir el tiempo necesario para el cálculo de pérdidas AC en un sistema SMES con una geometría arbitraria sometido a un transitorio cualquiera, sin comprometer la precisión de los resultados mediante la implementación de métodos numéricos en el lenguaje de programación Julia.

2. Definición del proyecto

El objetivo principal de esta tesis reside en desarrollar una simulación más eficiente para el cálculo de transitorios electromagnéticos en una bobina superconductora con geometría y transitorio arbitrarios, buscando una disminución significativa en el tiempo de ejecución respecto a la solución actual implementada en MATLAB.

El proceso completo consiste en la migración del código y los métodos de cálculo existentes en MATLAB al lenguaje Julia y su posterior optimización a nivel de software, modificando las funciones y cálculos, y a nivel de hardware, migrando aquellas secciones que suponen un cuello de botella a la GPU. La validación de los

resultados se realizará con un comparación con el modelo base en MATLAB, evaluando tanto la precisión de los resultados como el tiempo de ejecución. Se espera que el modelo resultante exhiba un desempeño superior al modelo en MATLAB, logrando una notable reducción en el tiempo de ejecución sin detrimento de la precisión de las pérdidas AC.

3. Descripción del modelo

Para comprobar el comportamiento del modelo se han escogido tres geometrías distintas: una primera geometría formada por dos bobinas conectadas en serie una encima de otra, en formato doble pancake, constituidas por 20 espiras de cinta superconductora (20,2); la segunda geometría consta de dos dobles pancakes formadas por 40 vueltas (40,4); y la última geometría es un doble pancake de 144 vueltas (144,2).

La Ilustración 1 muestra la geometría (20,2) del sistema SMES a simular sin escala. Todos los parámetros se mantienen para la segunda y tercera geometría excepto el número de bobinas, el número de espiras y el radio externo, que depende del número de vueltas de superconductor. Se han escogido tres geometrías arbitrarias para poder visualizar diferentes rendimientos de los modelos para complejidades incrementales del problema. El transitorio a simular se ha definido como una intensidad sinusoidal con una amplitud de 600 A a frecuencia 1 Hz durante 10 segundos.

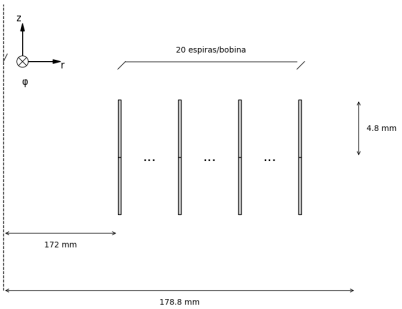


Ilustración 1 – Geometría (20,2) del sistema SMES.

El proceso de simulación del sistema SMES se segrega en 6 partes principales: Cargas y otros parámetros, Malla, Geometría, Condiciones de Contorno, Resolución y Post-Procesado. La implementación concreta con sus funciones principales viene detallada en el diagrama de flujo de la Ilustración 2. Las mejoras de software y de hardware se han centrado principalmente en la creación de matrices, pues abarca un porcentaje muy elevado del tiempo de computación.

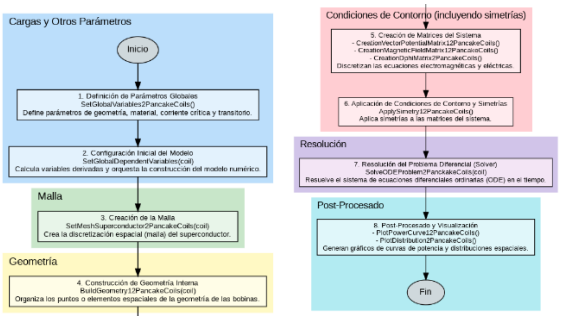


Ilustración 2 – Diagrama de Flujo Simulación SMES.

4. Resultados

El Modelo III representa la implementación total más optimizada, alcanzando una mejora superior al 60% en el tiempo de cálculo respecto al modelo de MATLAB para cualquier geometría. Esto significa que una tarea que en MATLAB tomaría aproximadamente 2253.26 segundos (alrededor de 38 minutos), por ejemplo, el Modelo III la resuelve en tan solo 144.29 segundos. Incluso para configuraciones más exigentes como (40,4) o (144,2) el Modelo III mantiene esta impresionante mejora superior al 60% sobre MATLAB, sin comprometer a penas la precisión del modelo, como se aprecia en las Ilustraciones 3 y 4.

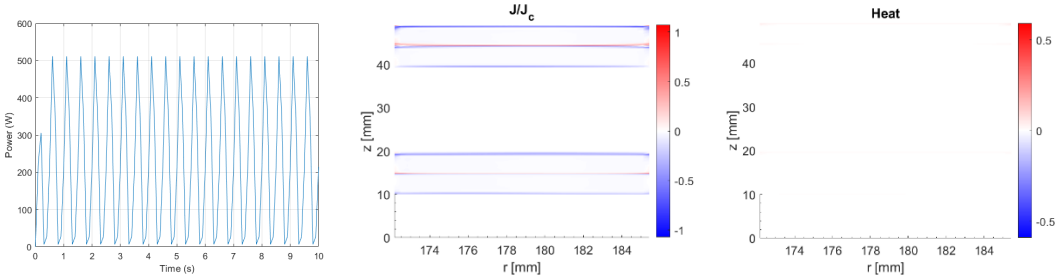


Ilustración 3 – Distribuciones en MATLAB de Potencia [W], J/Jc y Calor disipado para la Geometría (40,4).

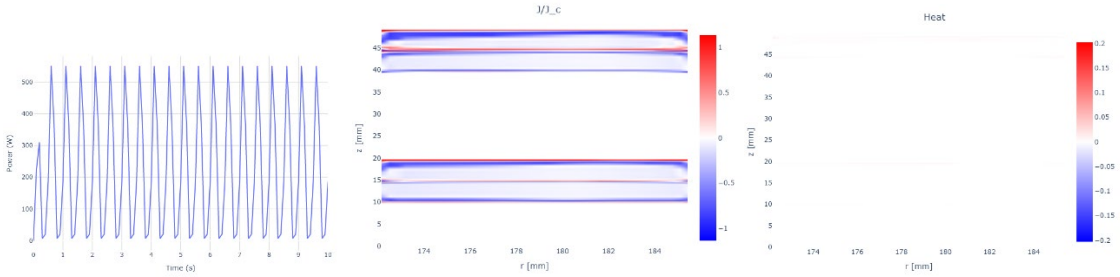


Ilustración 4 – Distribuciones en Julia con GPU optimizada (Modelo III) de Potencia [W], J/Jc y Calor disipado para la Geometría (40,4).

Geometría y Configuración	MATLAB (s)	Modelo III con solver (s)	Mejora (%)
(20,2)	2253.26	144.29	93.59
(40,4)	11930.70	3881.59	67.47
(144,2)	49622.90	19207.59	61.29

5. Conclusiones

Se ha construido de manera incremental un modelo computacional para simular el comportamiento de un sistema SMES en Julia, concentrándose prioritariamente en el cómputo de las matrices de potencial eléctrico y campo magnético, que junto con la resolución de las ecuaciones diferenciales del problema, constituyen la práctica totalidad del tiempo de cómputo. Independientemente de la complejidad de la geometría, se ha logrado una mejora global en el tiempo de computación de las matrices superior al 60% en comparación con el modelo base en MATLAB (desarrollado por J. Orea) [3] y el modelo final en Julia, optimizado para GPU y denominado Modelo III. Estas características hacen que el programa sea ideal para ser utilizado en la fase de diseño de la geometría del sistema SMES donde deben hacerse numerosas simulaciones con diversas geometrías y transitorios.

6. Referencias

[1] X. L. a. J. W. a. M. D. a. J. Clarke, «Overview of current development in electrical energy storage technologies and the application potential in power system operation,» *Applied Energy,* pp. 511-536, 2015.

[2] H. a. W. Z. a. G. F. a. G. K. a. M. M. Zhang, «Alternating Current Loss of Superconductors Applied to Superconducting Electrical Machines,» *Energies,* vol. 14, 2021.

[3] J. O. Rufino, MODELADO Y VALIDACIÓN DE PERDIDAS AC EN IMÁN SUPERCONDUCTOR DE ALTA TEMPERATURA, Trabajo Fin de Máster: Universidad Pontificia Comillas, 2024.

ADVANCED COMPUTATIONAL MODELING TECHNIQUES IN APPLIED SUPERCONDUCTIVITY

Author: Conderana Medem, Elena.
Supervisor: Hernando López de Toledo, Carlos José.
Collaborating Entity: Cyclomed Technologies

ABSTRACT

This study proposes a model for the computation of AC losses in a SMES system during electrical transients, implemented in Julia and optimized for GPU. The validation of the model against the benchmark implementation in MATLAB reveals an improvement in the computation of the electric potential and magnetic field matrices over 95%, as well as an improved convergence during the solution of differential equations. The final model simplifies the definition of the geometry and transients to be simulated, successfully streamlining the design process of a SMES system.
Keywords: SMES, HTS, AC losses, Julia, GPU, numerical simulation.

1. Introduction

SMES systems store energy in the form of a magnetic field [1]. For this purpose, current is circulated through a superconducting coil until a certain level is reached. Once this point is reached, the cryogenically cooled coil is short-circuited, storing the energy generated in the form of magnetic field indefinitely, since the zero resistivity of the superconductor does not generate current losses. However, one of the great challenges of this technology lies in the losses generated during alternating current (AC) charge and discharge cycles that increase the thermal load of the system [2]. To ensure optimal operation, keeping the temperature of the superconductor below the critical threshold, it is essential to simulate the electrical transients to measure the impact and monitor their state in real time, although this is a complex process with a high computational cost [2].

Faced with the challenge of computational time, this thesis aims to reduce the time required to calculate AC losses in a SMES system with an arbitrary geometry subjected to any transient, without compromising the accuracy of the results by implementing numerical methods in the Julia programming language.

2. Project definition

The main objective of this thesis is to develop a more efficient simulation for the calculation of electromagnetic transients in a superconducting coil with arbitrary geometry and transient, looking for a significant decrease in the execution time with respect to the current solution implemented in MATLAB.

The whole process consists of migrating the existing MATLAB code and computational methods to the Julia language and its subsequent optimization at software level, modifying functions and calculations, and at hardware level, migrating the sections that represent a bottleneck to the GPU. The validation of the results will be performed with a comparison with the benchmark model in MATLAB, evaluating both the accuracy

of the results and the execution time. The resulting model is expected to exhibit superior performance than the MATLAB model, achieving a notable reduction in execution time without detriment to AC loss accuracy.

3. Model description

Three different geometries have been chosen to test the behavior of the model: a first geometry formed by two coils connected in series one on top of the other, in double pancake format, consisting of 20 turns of superconducting tape (20,2); the second geometry consists of two double pancakes formed by 40 turns (40,4); and the last geometry is a double pancake of 144 turns (144,2).

Figure 1 shows the geometry (20,2) of the SMES system to be simulated without scaling. All parameters are kept for the second and third geometry except the number of coils, the number of turns and the external radius, which depends on the number of superconductor turns. Three arbitrary geometries have been chosen in order to visualize different performances of the models for incremental complexities of the problem. The transient to be simulated has been defined as a sinusoidal current with an amplitude of 600 A at frequency 1 Hz during 10 seconds.

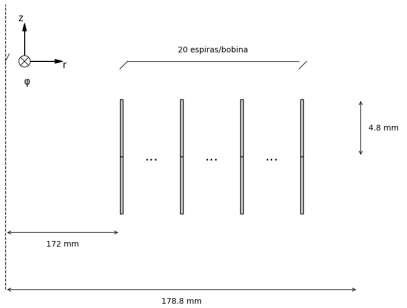


Figure 2 – Geometry (20,2) of SMES system.

The simulation process of the SMES system is divided into 6 main parts: Loads and other parameters, Mesh, Geometry, Boundary Conditions, Resolution and Post-Processing. The specific implementation with its main functions is detailed in the flowchart in Figure 2. The software and hardware improvements have been mainly focused on the creation of matrices, as it covers a very high percentage of the computational time.

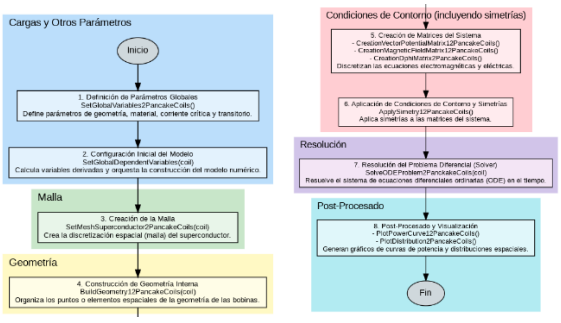


Figure 2 – Flowchart of SMES simulation.

4. Results

Model III represents the globally most optimized implementation, achieving over 60% improvement in computation time over the MATLAB model for any geometry. This means that a task that in MATLAB would take approximately 2253.6 seconds (about 38 minutes), for example, Model III solves it in only 144.29 seconds. Even for more demanding configurations such as (40,4) or (144,2) the Model III maintains that impressive 60%+ improvement over MATLAB, with almost no compromise in model accuracy, as shown in Figures 3 and 4.

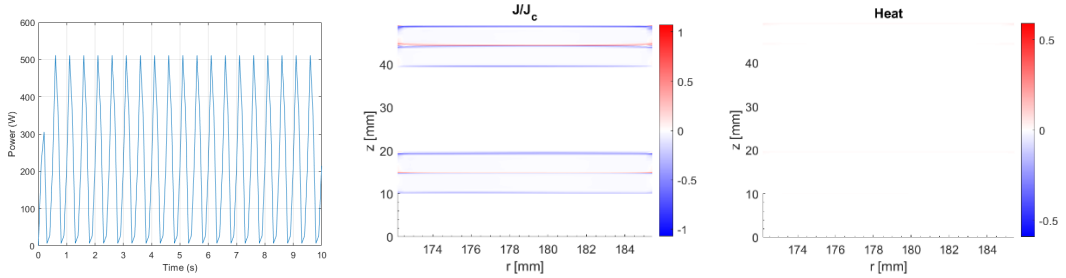


Figure 3 – MATLAB Distribution of Power [W], J/Jc and Heat dissipation for Geometry (40,4).

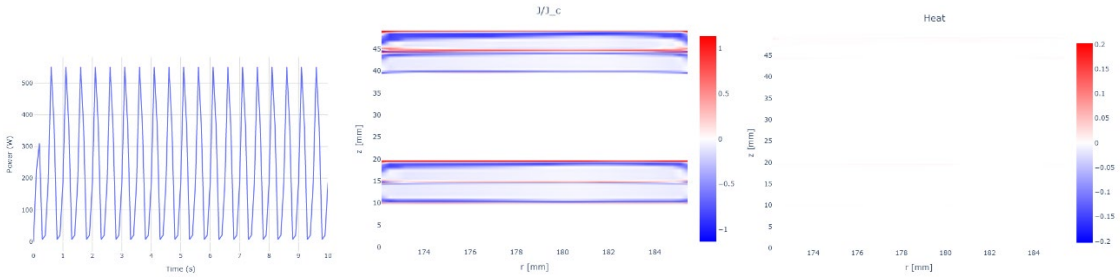


Figure 4 – GPU optimized Julia Distribution (Modelo III) for Power [W], J/Jc and Heat dissipation for Geometry (40,4).

Geometry and Configuration	MATLAB (s)	Model III with solver (s)	Improvement (%)
(20,2)	2253.26	144.29	93.59
(40,4)	11930.70	3881.59	67.47
(144,2)	49622.90	19207.59	61.29

5. Conclusions

A computational model has been incrementally built to simulate the behavior of a SMES system in Julia, concentrating mainly on the computation of the electric potential and magnetic field matrices, which together with the solution of the differential equations of the problem, constitute almost the total computation time. Regardless of the complexity of the geometry, an overall improvement in the computation time of the matrices over 60% has been achieved in comparison with the benchmark model in MATLAB (developed by J. Orea) [3] and the final model in Julia, optimized for GPU and called Model III. These features make the model ideal for use in the geometry design phase of the SMES system where numerous simulations with various geometries and transients must be performed.

6. References

[1] X. L. a. J. W. a. M. D. a. J. Clarke, «Overview of current development in electrical energy storage technologies and the application potential in power system operation,» *Applied Energy,* pp. 511-536, 2015.

[2] H. a. W. Z. a. G. F. a. G. K. a. M. M. Zhang, «Alternating Current Loss of Superconductors Applied to Superconducting Electrical Machines,» *Energies,* vol. 14, 2021.

[3] J. O. Rufino, MODELADO Y VALIDACIÓN DE PERDIDAS AC EN IMÁN SUPERCONDUCTOR DE ALTA TEMPERATURA, Trabajo Fin de Máster: Universidad Pontificia Comillas, 2024.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Existen ciertos materiales con la capacidad intrínseca de variar sus propiedades eléctricas y magnéticas a bajas temperaturas. Esta cualidad denominada superconductividad les permite conducir corriente eléctrica sin resistencia ni disipación de energía. Sin embargo, el uso de materiales superconductores ha estado limitado a aplicaciones concretas científicas y de medicina hasta el descubrimiento de superconductores de alta temperatura [1], HTS por sus siglas en inglés, cuyas temperaturas críticas por encima del punto de ebullición del nitrógeno líquido, 77K, permiten trabajar a temperaturas más manejables.

El aumento de la temperatura de operación junto con el descenso gradual del precio de los materiales ha convertido a los HTS en soluciones técnica y económicamente viables [2]. El uso de HTS abre la puerta a transformar el sector energético, evitando la pérdida de energía debido a la disipación en forma de calor causada por la resistencia de los materiales, contribuyendo así a un incremento de la eficiencia energética. Algunos de los desarrollos específicos llevados a cabo en este ámbito por empresas como CYCLOMED Technologies, con cuya colaboración se lleva a cabo esta tesis, comprenden SMES (Superconducting Magnetic Energy Storage), sistemas de almacenamiento de energía en forma de campo magnético que permiten almacenar corrientes eléctricas sin pérdidas [3], o generadores lineales para energía undimotriz.

Los sistemas SMES se componen de tres módulos principales, un subsistema de acondicionamiento de energía conectado a la red, un sistema de refrigeración y una bobina superconductora, que es el elemento central. La bobina se conecta a la red sin un estado de carga completo para aumentar la corriente que circula por ella misma hasta alcanzar una determinada intensidad. Alcanzado este punto la bobina enfriada criogénicamente se conecta en cortocircuito, almacenando de manera indefinida la energía generada en forma de campo magnético, pues la resistividad nula del superconductor no genera pérdidas de corriente. Para suministrar la energía almacenada y actuar como generador, la bobina se vuelve a conectar a

la red. La Figura 1.1 muestra un esquema simplificado de la arquitectura de un SMES.

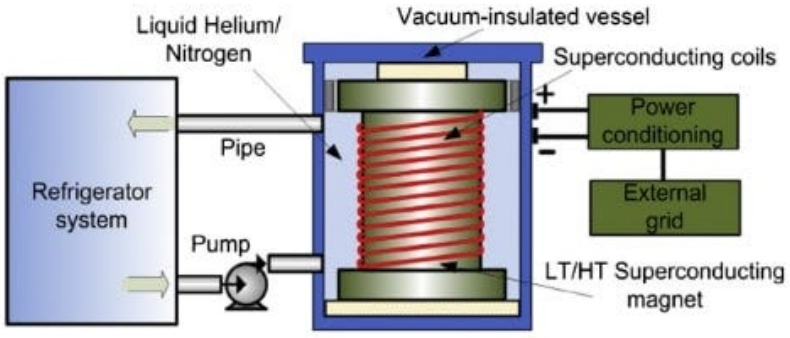


Figura 1.1: Estructura simplificada de un sistema SMES. [3]

La ventaja de esta tecnología frente a otras soluciones como baterías recargables o químicas reside en su almacenamiento de altas densidades de potencia (hasta 4000W/L), su velocidad de respuesta en cuestión de milisegundos y su capacidad de descarga inferior a un minuto. Además, ofrece una gran eficiencia de ciclos (95-98 %) y una larga vida útil (hasta 30 años). Estas cualidades han posicionado a esta tecnología como un opción competitiva para mitigar fluctuaciones de potencia de sistemas eólicos y fotovoltaicos, controlar la frecuencia, y mejorar la estabilidad transitoria y la calidad de la energía de la red ante fluctuaciones de carga elevadas, que requieren respuestas de potencia inmediatas [3]. Su aplicación también se extiende a buques militares o civiles en forma de baterías complementarias [4]

Sin embargo, esta tecnología enfrenta diversos desafíos. Uno de los principales reside en las pérdidas generadas durante los ciclos de carga y descarga en corriente alterna (AC) que incrementan la carga térmica del sistema [5]. Para garantizar el funcionamiento óptimo, manteniendo la temperatura del superconductor por debajo del umbral crítico, es fundamental la simulación de los transitorios electromagnéticos asociados a estos procesos para poder medir el impacto y monitorizar su estado en tiempo real, aunque se trata de un proceso complejo con un elevado coste computacional [5].

Ante la complejidad de la simulación de pérdidas AC, así como de otros retos en el diseño, operación y mantenimiento de superconductores, se hace evidente la importancia de emplear métodos de cálculo alternativos y técnicas avanzadas de modelado computacional. Un número creciente de estudios se ha inclinado por explorar el potencial de técnicas avanzadas de inteligencia artificial (IA), que han demostrado su eficacia en industrias con altas exigencias de fiabilidad y gestión de riesgos [1]. Para superar los altos tiempos de cálculo de modelos de elementos

finitos (FE) en simulaciones en tiempo real de sistemas superconductores, el uso de técnicas como redes neuronales artificiales (ANNs), árboles de decisión, K-nearest neighbour o XGBoost constituyen alternativas de bajo coste computacional para predecir el estado de un SMES en tiempo real, evaluando la seguridad operativa y calculando parámetros como pérdidas AC y temperatura final de la bobina HTS [1]. Asimismo, el uso de redes neuronales y técnicas de deep learning (DL) permiten realizar simulaciones más eficientes que los modelos analíticos y numéricos tradicionales, equilibrando precisión y velocidad para el cálculo de pérdidas AC y la optimización del diseño de componentes superconductores [1].

Otros estudios abogan por seguir mejorando los métodos numéricos tradicionales, pero implementando formulaciones matemáticas alternativas para facilitar su resolución en softwares como MATLAB o plataformas gratuitas como Julia [6]. Aunque el tiempo de computación entre algunos solvers para ecuaciones diferenciales es comparable entre ambos entornos [6], la generación de las matrices de campo eléctrico y magnético en MATLAB, a partir de formulaciones integrales complejas, requiere largos periodos de computación.

Ante el desafío que representa el tiempo de cómputo, esta tesis tiene como objetivo reducir dicho tiempo necesario para el cálculo de pérdidas AC en un sistema SMES con una geometría arbitraria sometido a un transitorio cualquiera, sin comprometer la precisión de los resultados mediante la implementación de métodos numéricos en el lenguaje de programación Julia. Un lenguaje desarrollado para proporcionar una resolución eficiente de problemas físicos y numéricos mediante la paralelización y la vectorización de sus códigos. El trabajo se fundamenta sobre la investigación de Javier Orea [7], que servirá como benchmark para comparar el rendimiento y la precisión entre el modelo de MATLAB y el nuevo desarrollo en Julia.

1.1. Objetivos

El objetivo principal de esta tesis reside en desarrollar una simulación más eficiente para el cálculo de transitorios electromagnéticos en una bobina superconductora con geometría y transitorio arbitrarios, buscando una disminución significativa en el tiempo de ejecución respecto a la solución actual implementada en MATLAB.

El primer paso consiste en la migración del código y los métodos de cálculo existentes en MATLAB al lenguaje de programación Julia, replicando su funcionalidad y optimizando el código mediante su vectorización. La validación de los resultados se realizará con un comparación con el modelo base en MATLAB, evaluando tanto la precisión de los resultados como el tiempo de ejecución.

Posteriormente, se procederá a la optimización del modelo en Julia a través

de mejoras a nivel de software, refinando los cálculos existentes e implementando modificaciones estructurales en las funciones para simplificar procesos y mejorar la eficiencia sin comprometer la precisión. Además, se explorará la aceleración mediante hardware, trasladando a la GPU las secciones del código que representen cuellos de botella y consumen la mayor parte de los recursos computacionales.

La conclusión del proyecto consistirá en la validación experimental de la metodología de cálculo desarrollada, evaluando empíricamente su precisión y rendimiento. Se espera que el modelo resultante exhiba un desempeño superior al modelo en MATLAB, logrando una notable reducción en el tiempo de ejecución sin detrimento de la precisión en el cálculo de pérdidas AC.

1.2. Motivación

El diseño preliminar de un SMES implica un balance entre múltiples aspectos, donde destacan la máxima energía magnética almacenada, el peso del sistema y las pérdidas AC generadas durante los transitorios eléctricos, así como determinar sus parámetros geométricos incluyendo el número de bobinas o el radio interno de cada una. Este proceso iterativo requiere numerosas simulaciones con diversas geometrías y escenarios de transitorios para alcanzar un compromiso óptimo entre los distintos aspectos críticos [7]. Además, aunque estos sistemas pueden generar campos magnéticos intensos, también están sujetos a limitaciones significativas, como el quench electrotérmico. Este fenómeno ocurre cuando la corriente en el superconductor excede su corriente crítica provocando un rápido aumento de la temperatura que puede llegar a dañar permanentemente la bobina HTS [8].

Dada la complejidad de los fenómenos térmicos y electromagnéticos, y la importancia de cuantificarlos e identificarlos, existen desde métodos numéricos tradicionales basados en métodos de elementos finitos (FEM) y métodos de diferencias finitas (FDM), hasta softwares comerciales consolidados en la industria, como COMSOL Multiphysics y ANSYS, o soluciones implementadas en entornos como MATLAB que han abordado la simulación de estos transitorios. Sin embargo, todos estos enfoques comparten una limitación común: la velocidad de convergencia en el análisis electromagnético suele ser considerablemente lenta [8]. Incluso MATLAB, que destaca en el manejo eficiente de operaciones matriciales y vectoriales introduce una pérdida significativa en el rendimiento computacional debido a la necesidad de recurrir a estructuras de bucles for para abordar la complejidad inherente a ciertas geometrías, generando cuellos de botella importantes que aumentan considerablemente los tiempos de simulación [7].

Esta limitación motiva la exploración de métodos numéricos alternativos y el desarrollo de nuevas herramientas de cálculo rápidas y robustas, que ofrezcan resultados con un nivel de precisión aceptable y que permita la modificación sencilla

de la geometría del sistema bajo estudio. En este contexto la adopción de lenguajes como Julia con una mayor eficiencia computacional ofrece la ventaja de poder realizar un mayor número de simulaciones en un menor lapso de tiempo. Esto facilita la exploración de un espacio de diseño más amplio y la optimización de los sistemas SMES. Al poder llevar a cabo simulaciones con una mayor resolución espacial y temporal, se potencia la obtención de resultados más precisos en el mismo o incluso menor tiempo que simulaciones más simples.

En este contexto, este proyecto se orienta al desarrollo de un programa en Julia con velocidades altas de convergencia ofreciendo resultados rápidos, con un nivel de precisión adecuado. Asimismo, el modelo debe permitir la modificación sencilla de la geometría del sistema, con el objetivo final de agilizar la determinación de la geometría óptima y poder comprobar el estado de un sistema SMES.

1.3. Metodología

La ejecución de la tesis se divide en 4 fases principales. La primera fase consiste en la revisión de literatura y de métodos numéricos para familiarizarse con la superconductividad, en concreto los HTS y los SMES, así como con las soluciones actuales para la determinación de los transitorios electromagnéticos y en concreto con el modelo desarrollado en MATLAB por Javier Orea (Véase: [7]).

La segunda fase comprende la familiarización con el lenguaje de programación Julia, diseñado por el MIT para la computación numérica de alto rendimiento siendo capaz de alcanzar velocidades comparables a lenguajes de bajo nivel como C o C++, pero conservando la sencillez en la sintaxis propia de lenguajes de alto nivel como Python. A pesar de la similitud de la sintaxis con Python, es necesario profundizar en la vectorización y paralelización características de este lenguaje, pues son los elementos claves para lograr en la migración desde MATLAB un significativo incremento en la eficiencia computacional.

Concluida la migración y adaptación inicial se procederá a la optimización del modelo a través de mejoras en los cálculos existentes y modificaciones estructurales en las funciones para simplificar procesos y mejorar la eficiencia sin comprometer la precisión. Además, se explorará el traslado a la GPU de las secciones del código que representen cuellos de botella y consumen la mayor parte de los recursos computacionales.

Por última, la fase terminal consiste en la comprobación empírica de la bondad del modelo desarrollado y en la comparación del consumo de recursos frente a otras soluciones similares, principalmente con su modelo base en MATLAB, para establecer si el modelo desarrollado consigue abordar exitosamente el problema propuesta.

Capítulo 2

Fundamentos Matemáticos y Estado de la Cuestión

2.1. Revision de las distintas metodologias para el analisis de transitorios electromagneticos, y el calculo de perdidas AC

Todas las aplicaciones de superconductores, desde los sistemas de resonancia magnética nuclear (RMN) hasta el diseño de SMES, requieren un modelado electromagnético preciso. Generalmente estas aplicaciones operan con corrientes o campos magnéticos con frecuencias características que varían desde mHz o incluso inferiores, como en el caso de los imanes superconductores, hasta los 50-60 Hz utilizados en dispositivos para redes eléctricas [9]. En cualquier caso, la frecuencia es lo suficientemente baja como para despreciar la aparición de corrientes de desplazamiento, lo cual permite trabajar bajo la aproximación de campos magnéticos que varían lentamente. Bajo esta presunción se puede considerar que todo el espacio de interés está gobernado por las ecuaciones de Maxwell de cuasi-magnetoestática:

$$\nabla \cdot \mathbf{D} = q, \qquad \text{Gauss'law,} \tag{2.1}$$

$$\nabla \cdot \mathbf{B} = 0, \qquad \text{magnetic flux conservation law,} \tag{2.2}$$

$$\nabla \times \mathbf{E} = -\partial_t \mathbf{B}, \qquad \text{Faraday's law,} \tag{2.3}$$

$$\nabla \times \mathbf{H} = \mathbf{J}, \qquad \text{Ampère's law,} \tag{2.4}$$

donde, D es el desplazamiento eléctrico, B es el campo magnético, E es el campo eléctrico, H es la intensidad de campo magnético, J es la densidad de corriente, y q es la densidad de carga. Para resolver las ecuaciones ((2.1)–(2.4)) se deben imponer las siguientes ecuaciones constitutivas del material:

$$D = \varepsilon \cdot E \tag{2.5}$$

$$B = \mu \cdot H \tag{2.6}$$

donde, ε es la permitividad, μ es la permeabilidad y ρ la resistividad del material, aunque para determinar las pérdidas AC la permitividad y la permeabilidad del superconductor pueden definirse como la del aire [2]. En las regiones superconductoras, sin embargo, la propiedad más importante es la resistividad, definida por la siguiente expresión ampliamente utilizada en el modelado de superconductores [9]:

$$\rho = E_c \cdot \left| \frac{J}{J_c} \right|^{n-1} \tag{2.7}$$

donde, el exponente n, J_c, la densidad de corriente crítica que puede conducir una corriente eléctrica sin disipación de energía [10], y E_c el campo eléctrico crítico generado por la densidade de corriente crítica, dependen del material superconductor y definen su comportamiento. La densidad de corriente crítica depende de la temperatura T, del módulo del campo magnético $|\mathbf{B}|$ y del ángulo de incidencia del campo magnético respecto a la superficie del superconductor θ [11]. La Figura 2.1 representa dicha dependencia para una cinta (RE)BCO a 4.2 K y para un campo de 1 T y 30 T respectivamente.

$$J_c = f(T, |\mathbf{B}|, \theta) \tag{2.8}$$

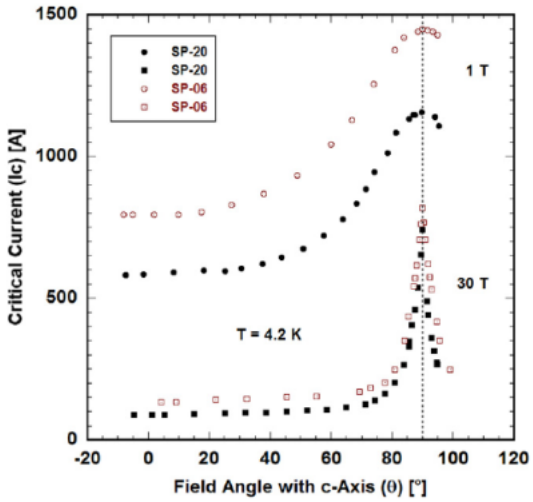


Figura 2.1: Corriente crítica en función de T, B y θ [11].

Por último, el carácter altamente no lineal de los superconductores plantea un desafío en la predicción del calor generado, expresado como:

$$Q = \int_0^t \left(\iiint_V \mathbf{E} \cdot \mathbf{J} \, dV \right) dt \tag{2.9}$$

donde, Q es el calor total generado en el volumen V del superconductor desde el instante inicial hasta el tiempo t. Debido a que las pérdidas AC en superconductores HTS constituyen un grave problema para la refrigeración, incrementando el coste operacional de aplicaciones superconductoras y limitando su eficiencia, la resolución de este problema ha sido ampliamente estudiada. Existen desde métodos numéricos que permiten modelar geometrías y situaciones más complejas hasta métodos analíticos que son más eficientes para computar las pérdidas de geometrías de HTS simples. De hecho, es posible combinar distintas formulaciones en un mismo problema aplicando cada una a la región del espacio de interés donde mejor se adapte [9].

Dentro de los métodos numéricos existen distintos enfoques: métodos basados en elementos finitos, métodos variacionales (que encuentran variables electromagnéticas mediante la minimización de funciones), métodos integrales y métodos espectrales [9]. Puesto que este trabajo se centra en métodos basados en elementos finitos, se van a analizar a continuación en mayor profundidad algunas de las soluciones más generalizadas de este enfoque.

2.1.1. Formulación-H

La formulación-H, cuyo nombre se deriva de su variable dependiente, la intensidad de campo magnético **H**, es uno de los modelos de topologías HTS más difundidos [9]. Destaca por su precisión y convergencia para simular distintas topologías y aplicaciones, así como por su facilidad de implementación en softwares de elementos finitos como COMSOL Multiphysics [2]. Este método expresa el cálculo de pérdidas AC en función de las variables **H** y **J** mediante la resolución por elementos finitos de la ecuación de Faraday expresada en términos de campo magnético, modelando el material superconductor como un material con una permeabilidad magnética relativa μ_r:

$$\nabla \times (\rho \nabla \times \mathbf{H}) + \mu_0 \mu_r \frac{\partial \mathbf{H}}{\partial t} = 0 \tag{2.10}$$

La densidad de corriente **J** se deriva del campo magnético mediante la aproximación quasi-estática de la ley de Ampère [2]:

$$\mathbf{J} = \nabla \times \mathbf{H} \tag{2.11}$$

Para resolver la ecuación diferencial de campo magnético es necesario añadir condiciones frontera y restricciones necesarias para adaptar la formulación al caso específico de uso. La siguiente expresión sería necesaria si se quisiera incluir una restricción para definir la corriente I que debe circular a través de una superficie [9]:

$$I = \oint_{\partial S} \mathbf{H} \cdot d\mathbf{l} \tag{2.12}$$

donde, ∂S hace referencia al contorno de superficie y dl a un elemento diferencial de su longitud.

Una de las principales limitaciones de esta formulación es el alto tiempo y carga de cómputo en escenarios con numerosas cintas conductoras o gran cantidad de número de espiras, siendo hasta 20 veces más lento que la formulación T-A [2], una de las alternativas propuestas para enfrentar estos problemas.

2.1.2. Formulación T-A

La formulación T-A desprecia el espesor de la cinta superconductora HTS, centrándose en resolver el vector potencial de corriente $\mathbf{T}$ sólo en el dominio superconductor, y el vector potencial magnético $\mathbf{A}$ para el dominio completo. Ambas variables estado, $\mathbf{T}$ y $\mathbf{A}$, se emplean para calcular la densidad de corriente $\mathbf{J}$ y la densidad de flujo magnético $\mathbf{B}$ respectivamente. El método resuelve de manera simultánea tanto $\mathbf{J}$ como $\mathbf{B}$ intercambiando estas variables continuamente [9]. Las ecuaciones que gobiernan este método en su forma más genérica son:

$$\mathbf{J} = \nabla \times \mathbf{T} \tag{2.13}$$

$$\mathbf{B} = \nabla \times \mathbf{A} \tag{2.14}$$

A partir de las cuales se derivan las siguientes relaciones y aplicaciones de leyes de Maxwell:

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{2.15}$$

$$\mathbf{E} = \rho_{HTS}\mathbf{J} \tag{2.16}$$

$$\nabla \times \mathbf{H} = \mathbf{J} + \mathbf{J}_s \tag{2.17}$$

$$\mathbf{B} = \mu_0 \mathbf{H} \tag{2.18}$$

$$\mathbf{J}_s = \mathbf{J}\delta \tag{2.19}$$

donde, ρ es la resistividad del material HTS, μ_0 y μ_r representan la permeabilidad magnética del vacío y relativa respectivamente, y $\mathbf{J}_s$ es la densidad de corriente externa superficial aplicada en A/m y δ el espesor de la cinta HTS. Cabe

destacar que la última ecuación desprecia el espesor de la capa superconductora del material HTS utilizando la aproximación de lámina delgada [12].

Como ocurre en la formulación-H, es necesario imponer las condiciones de contorno y restricciones correspondientes al problema a resolver. En esta formulación la restricción para definir la corriente I que debe circular a través de una superficie S sería:

$$\mathbf{I} = \iint_S \mathbf{J} \cdot d\mathbf{S} = \oint_{\partial S} \mathbf{T} \cdot d\mathbf{l} \tag{2.20}$$

donde, dS hace referencia a un diferencial de la superficie S.

Aunque la formulación T-A es más eficiente que la formulación-H, debido a la computación simultánea de dos variables y al intercambio continuo de la densidad de corriente $\mathbf{J}$ y la densidad de flujo magnética $\mathbf{B}$, su uso está limitado a casos donde la aproximación de lámina delgada de la capa superconductora es significativa, lo cual no puede realizarse con geometrías complejas [12].

Aunque ambas formulaciones muestran una gran capacidad para abordar el problema de las pérdidas AC, la siguiente sección profundiza en el trabajo desarrollado por J. Orea para reducir el tiempo de ejecución de las formulaciones-H y T-A, permitiendo además, calcular los transitorios electromagnéticos para bobinas superconductoras con geometrías y transitorios arbitrarios.

2.2. Revision exhaustiva de metodologias y metodos de calculo existentes de transitorios EMs

El proyecto de J. Orea consiste en el desarrollo de un modelo para las pérdidas AC generadas en una bobina superconductora durante transitorios eléctricos. El objetivo de este modelo reside en agilizar el proceso de diseño de la geometría de un sistema SMES, superando las prestaciones de programas como COMSOL Multiphysics mediante mejoras en el tiempo de ejecución y en la facilidad de definir geometrías y transitorios a simular. El modelo se ha desarrollado basándose en el modelo integral planteado por Simon Otten y Francesco Grilli [6], algunas de cuyas ventajas comprenden:

- **Menor consumo de tiempo y de memoria:** al resolver solo corrientes en los conductores, el sistema de ecuaciones es mucho más pequeño que en métodos PDE volumétricos.

- **Sin mallar aire ni aislamiento:** desaparece la necesidad de expandir el dominio y definir condiciones de contorno artificiales; solo se discretizan las cintas superconductoras.
- **Ideales para cintas delgadas:** representar cada cinta como una lámina de corriente evita elementos FEM de alto aspecto que ralentizan la convergencia.
- **Escalabilidad a imanes grandes:** el tamaño del problema crece con el número de conductores, no con el volumen circundante; permite simular miles de espiras de forma tractable.
- **Condición de contorno abierta "natural":** el núcleo de Biot-Savart satisface Maxwell en el vacío, por lo que el campo lejano se calcula con exactitud sin trucos de frontera infinita.
- **Implementación sencilla y acoplamiento directo a circuitos:** se reduce a resolver ODEs de tensiones inductivas, fácilmente programables y reutilizables en modelos de protección o electrónica de potencia.
- **Útiles para estudios paramétricos y diseño rápido:** la velocidad de cálculo favorece iteraciones de optimización y análisis de pérdidas AC en fases tempranas del diseño.

2.2.1. Fundamentos Físicos y Matemáticos

J. Orea desarrolla un modelo en Matlab basado en el planteamiento de Simon Otten y Francesco Grilli [6], aplicando algunas modificaciones para adaptar el planteamiento a la geometría de estudio, de manera que las pérdidas AC se calculan considerando la forma de la distribución de corriente en cada espira. Dada la simetría del problema cada espira se divide en el mismo número N de elementos finitos con simetría de revolución de sección rectangular en la dirección z, como se aprecia en la Figura 2.2. En la dirección r no se realiza esta división debido al reducido espesor de la cinta. Cabe resaltar que la división en elementos finitos se ha relizado en tamaño descendente conforme se aproximan a los bordes para aumentar la precisión ante el efecto borde de las espiras.

Para considerar la distribución de densidad de corriente, se puede aplicar la ecuación general de electromagnetismo en cada elemento finito, que relaciona el vector potencial magnético A con el campo eléctrico E y el potencial escalara eléctrico $\nabla\phi$, donde el vector potencial magnético se divide de A_{ind}, que representa el potencial vector magnético inducido por todos los elementos de esa espira incluyendo el que se está evaluando, y A_{ext}, que representa el vector potencial magnético debido a los elementos finitos del resto de espiras:

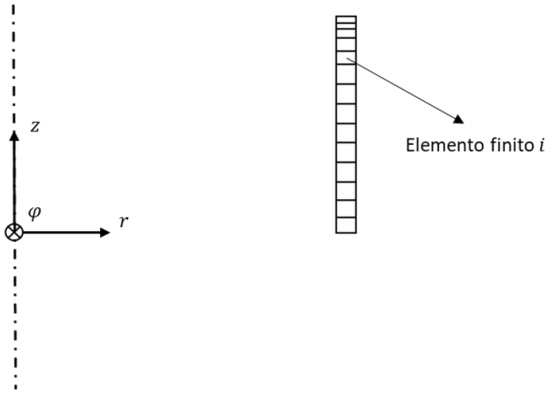


Figura 2.2: Elementos finitos de una espira. [7]

$$\frac{\partial \mathbf{A}_{ind}}{\partial t} = -\frac{\partial \mathbf{A}_{ext}}{\partial t} - \mathbf{E} - \nabla\phi \tag{2.21}$$

El desarrollo de las ecuaciones de Maxwell de magnetoestática dan lugar al siguiente cálculo de vector potencial magnético:

$$\mathbf{A}(\mathbf{r})\hat{\boldsymbol{\phi}} = \frac{\mu}{4\pi}\iiint_V \frac{\mathbf{J}\hat{\boldsymbol{\phi}}}{|\mathbf{r}-\mathbf{r}'|}dV \tag{2.22}$$

donde, V representa el volumen de todo el espacio, r' el vector posición desde el sistema de coordenadas a un punto dentro del volumen V, $\hat{\boldsymbol{\phi}}$ el vector unitario azimutal y r el vector posición donde se quiere evaluar el potencial vector magnético. Es necesario sustituir la derivada temporal del vector potencial magnético externo por el que generaría la distribución de corriente del resto de espiras sobre el elemento finito i contenido en la espira e:

$$\frac{\partial A_{ext_{ei}}}{\partial t} = \sum_{\substack{k=1 \\ k\neq e}}^{E}\left(K_{kij}\frac{dJ_{ej}}{dt}\right) \tag{2.23}$$

donde, E es el número de espiras totales. Además, el potencial escalar eléctrico de una espira por la que circula una corriente solo varía en la dirección de la intensidad, motivo por el cual se impone la siguiente restricción para que por cada espira de la bobina circule la intensidad del transitorio:

$$\nabla\phi_e = \gamma\left(I(t) - \sum_{i=1}^{N} J_{ei}dS_{ei}\right) \tag{2.24}$$

donde, dS_{ei} representa el área de la sección del elemento finito i en la espira e, $I(t)$ es la intensidad en función del tiempo que se quiere imponer, y γ es una

constante que debería ser lo suficientemente grande como para anular el término entre paréntesis, que representa el error entre la intensidad que se desea imponer y la real, que idealmente debería ser cero.

La sustitución de las funciones ((2.23)–(2.24)) en 2.21 dan lugar a la siguiente función 2.25, que se aplica a cada elemento finito (i) en los que se divide cada espira (e) considerando el impacto del resto de espiras (k) para cuantificar la distribución de densidad de corriente:

$$\sum_{\substack{k=1 \\ k\neq e}}^{E}\sum_{j=1}^{N}\left(K_{kij}\frac{dJ_{ej}}{dt}\right) = -\left(E_c\frac{J_{ei}}{J_{c_{ei}}}\left|\frac{J_{ei}}{J_{c_{ei}}}\right|^{n-1} + \gamma\left(I(t) - \sum_{i=1}^{N} J_{ei}dS_{ei}\right)\right) \tag{2.25}$$

Se trata de un sistema de ecuaciones diferenciales con tantas ecuaciones como incógnitas, donde las incógnitas son las densidades de corriente de cada elemento de cada espira $J_{e,i}$. Cabe resaltar el cálculo de $K_{e,i}^{k,j}$, que conlleva la mayor parte de la carga computacional. Este término depende exclusivamente de la geometría del problema y su ecuación toma la forma:

$$K_{kij} = \frac{\mu}{4\pi}\iiint_{V_{kj}} \frac{1}{|r_{ki} - r'_{kj}|}dV \tag{2.26}$$

donde V_j es el volumen del elemento finito j perteneciente a la espira k, $\mathbf{r}_{ki}$ es el vector posición desde el sistema de coordenadas hasta el elemento finito i, y $\mathbf{r}_{kj}$ es el vector posición desde el sistema de coordenadas al punto medio del volumen V_j. Las integrales triples se calculan en Matlab de manera numérica, resolviendo la integral respecto de la coordenada axial analíticamente y respecto de la coordenada radial empleando la regla de 1/3 de Simpson, concluyendo con la integración numérica entorno a la coordenada azimutal [7].

Para poder determinar cómo evolucionan las corrientes en el sistema Jc, es necesario utilizar la expresión que describe el campo magnético en todos los puntos espaciales y temporales, pues J_c depende de T, B y θ. Para calcular el campo magnético en cada elemento finito se emplea la Ley de Biot-Savart, una solución general que se obtiene al resolver las ecuaciones de Maxwell de magnetoestática:

$$\mathbf{B}(\mathbf{r}_{ei}) = \frac{\mu}{4\pi}\iiint_{V_{ej}} \mathbf{J}_{ei} \times \frac{(\mathbf{r}_{ei} - \mathbf{r}'_{ej})}{|\mathbf{r}_{ei} - \mathbf{r}'_{ej}|^3}dV \tag{2.27}$$

donde, $B(r_{ei})$ es el vector campo magnético en el elemento i perteneciente a la espira e, V_{ej} es el volumen del elemento finito j perteneciente a la espira e, r_{ei} es el vector posición desde el sistema de coordenadas hasta el punto medio del elemento i de la espira e, y r'_{ej} es el vector posición desde el sistema de coordenadas al punto

medio del volumen V_{ej}. De esta ecuación 2.27, se concluye que el campo magnético que experimenta el elemento i de la espira e viene dado por:

$$\mathbf{B}_{ei} = \sum_{\substack{k=1 \\ k \neq e}}^{E} \sum_{j=1}^{N} \mathbf{M}_{kij} J_{ej} \tag{2.28}$$

donde,

$$\mathbf{M}_{kij} = \frac{\mu}{4\pi} \iiint_{V_{ej}} \hat{\phi} \times \frac{(\mathbf{r}_{ei} - \mathbf{r}'_{ej})}{|\mathbf{r}_{ei} - \mathbf{r}'_{ej}|^3} dV \tag{2.29}$$

Las integrales triples se calculan numéricamente en Matlab mediante la implementación de bucles for. Cabe resaltar que para los valores en los que r_{ei} y r'_{ej} coinciden se ha fijado un valor nulo, pues la ley de Biot-Savart no es aplicable en dichos casos.

El sistema de ecuaciones resultante, por lo tanto, es de la forma:

$$[K]\frac{d[J]}{dt} = f(t, [J]) \tag{2.30}$$

donde, $[K]$ es la matriz de términos $K_{e,i}$ y [J] la matriz de incógnitas compuestas por las densidades de corriente de cada elemento, dando lugar a un sistema de ecuaciones igual al número de incógnitas que se resuelve mediante el uso de *solvers* de ODEs en Matlab.

2.2.2. Modelo Base en MATLAB

La formulación matemática explicada en la sección 2.2.1 se materializa en un modelo de simulación de un sistema SMES implementado en MATLAB, un entorno de programación y software matemático ampliamente utilizado en la comunidad científica y académica. MATLAB ofrece un entorno de desarrollo integrado con su propio lenguaje interpretado, caracterizado por una sintaxis de alto nivel diseñado para la manipulación de matrices, la implementación de algoritmos y la representación gráfica de funciones. Además, su ecosistema incluye una amplia colección de herramientas y paquetes que expanden sus capacidades, facilitando el prototipado rápido y la visualización inmediata de resultados [13].

El proceso de simulación del sistema SMES se segrega en 6 partes principales: Cargas y otros parámetros, Malla, Geometría, Condiciones de Contorno, Resolución y Post-Procesado. La implementación concreta con sus funciones principales viene detallada en el diagrama de flujo de la Figura 2.3. Aunque no se siga el orden habitual para simulaciones físicas, se ha optado por mostrar la secuencia seguida por el código.

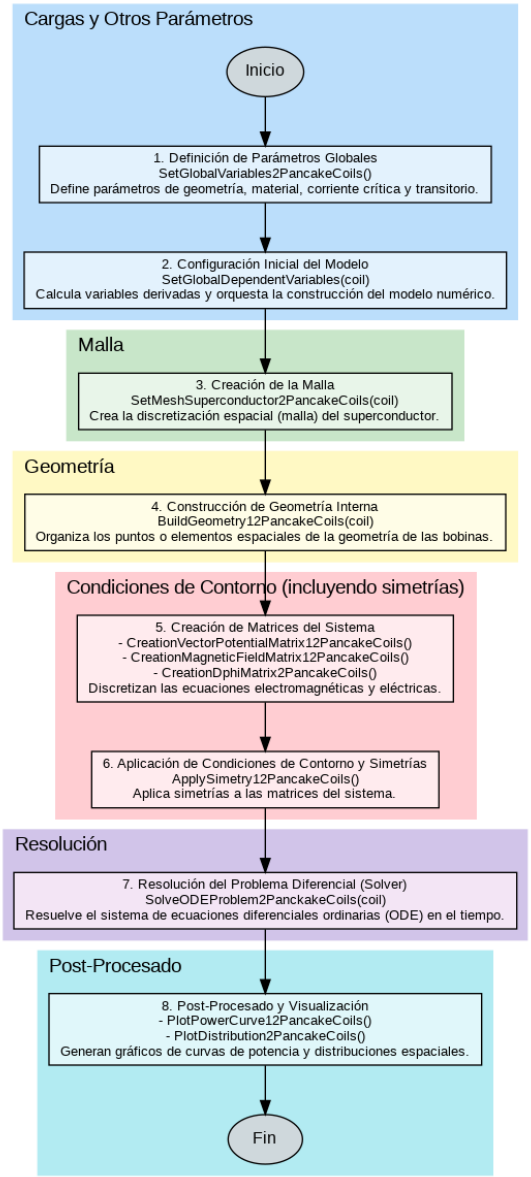


Figura 2.3: Diagrama de Flujo Simulación SMES.

La función `SetMeshSuperconductor2PancakeCoils` se encarga de la discretización espacial de la geometría de las bobinas, definiendo la malla (o *mesh*) que representa el volumen de las bobinas en un sistema de coordenadas cilíndricas.Para ello es necesario la creación de elementos discretos especificando sus límites en las dimensiones radial (r) y axial (z), y el cálculo de propiedades como el área de la sección transversal (`s`) y el volumen (`V`) de cada elemento. `BuildGeometry12PancakeCoils` se ocupa de la consolidación de los datos geométricos previamente generados organizando y estructurando los vectores de coordenadas (`r_low`, `r_middle`, `r_high`, `z_low`, `z_middle`, `z_high`) en formatos adecuados para las siguientes etapas de cálculo. Esencialmente, la funcionalidad de estas dos funciones reside en preparar los datos geométricos para su uso en la formulación de las matrices necesarias para la resolución del problema.

`CreationVectorPotentialMatrix12PancakeCoils` es una función crucial, pues es la responsable de construir la matriz del potencial vectorial (`K`). Esta matriz es fundamental en los métodos de elementos finitos para la resolución de problemas electromagnéticos, siendo la exactitud y eficiencia en su formación determinantes para la convergencia y la velocidad de la simulación global como se aprecia en la Tabla 2.2. La función calcula los coeficientes de interacción magnética entre los diferentes elementos de la malla mediante la evaluación de integrales complejas. Complementariamente, `CreationMagneticFieldMatrix12PancakeCoils` se encarga de generar las matrices de campo magnético (`KBr` y `KBz`) en las direcciones radial y axial respectivamente deduciendo y evaluando las expresiones de campo magnético generadas por cada elemento diferencial de corriente sobre los demás puntos de la malla. Estas matrices son esenciales para determinar la distribución del campo magnético resultante en el sistema.

La función `CreationDphiMatrix2PancakeCoils` es responsable de construir la matriz de superficie (`S`) y su comportamiento se detallará en más profundidad en la sección 3.1. Finalmente, `ApplySymmetry12PancakeCoils` se utiliza para aplicar reducciones basadas en las propiedades de simetría del sistema, cuando estas son aplicables. Su objetivo es optimizar el tamaño de las matrices (`K`, `KBr`, `KBz` y `S_super`) al explotar las redundancias inherentes a geometrías simétricas, lo cual permite una reducción significativa del coste computacional y de la memoria requerida.

El código original en MATLAB convierte directamente las ecuaciones físicas y las integrales previamente descritas 2.2.1 en algoritmos computacionales. Sin embargo, dada la complejidad de las integrales necesarias para el cálculo de la densidad de corriente eléctrica y del campo magnético, el desarrollo inicial del modelo se basa en un extenso uso de bucles anidados para iterar sobre los elementos discretos de las bobinas y realizar las integraciones numéricas. Esta implementación se traduce en la predicción de la distribución de densidad de corriente a

lo largo del tiempo con una precisión entre el 98 % y el 99 %, además de predecir razonablemente bien la distribución del campo magnético y la distribución de potencia. No obstante, el modelo enfrenta desafíos de rendimiento a causa de la implementación directa mediante bucles anidados de las integrales numéricas.

Limitaciones de Rendimiento

Aunque el tiempo de ejecución del modelo de MATLAB se reduce en un orden de magnitud frente a métodos que emplean softwares como COMSOL Multiphysics y los métodos numéricos explicados en la sección 2.1, se sigue tratando de un modelo con limitaciones en eficiencia computacional, manifestadas en tiempos de ejecución excesivamente largos para simulaciones de mayor escala, que reflejan aplicaciones reales. Esta ineficiencia se muestra en la sección 2.2.4. El tiempo de computación es principalmente consumido por el cálculo de los términos de la matriz $[K]$ y $[M]$, que dependen exclusivamente de la geometría del problema, si bien cabe mencionar que dichos valores están sujetos a ser reutilizados si se desea simular otro transitorio con la misma geometría, lo cual daría lugar a una simulación casi inmediata. Las limitaciones de rendimiento, especialmente para simulaciones complejas, se atribuyen principalmente a:

- **Falta de Vectorización:** La implementación en MATLAB hace un uso intensivo de bucles y bucles anidados, dando lugar a mayores tiempos de compilación para cálculos que vectorizando podrían incrementar su eficiencia. Además, el uso de bucles anidados para el cálculo de integrales dobles y triples incrementa los cuellos de botella que suponen el cómputo de $[K]$ y de $[M]$.

- **Falta de Paralelismo:** Aunque MATLAB ofrece capacidades de paralelización, el código realiza los cálculos de manera secuencial en la CPU, resultando en tiempos de ejecución muy elevados conforme aumenta el número de bobinas N_e o de espiras N_c en el modelo.

- **Cálculos Redundantes e Ineficiencias a Bajo Nivel:** El análisis exhaustivo del código y de la geometría del problema revelan la existencia de ineficiencias computacionales en algunos cálculos y funciones. Ciertos valores intermedios son recalculados repetidamente dentro de los bucles de integración, en lugar de ser almacenados y reutilizados. Además algunos cálculos de matrices y términos no contribuyen al resultado final, incrementando innecesariamente la carga y tiempo de computación.

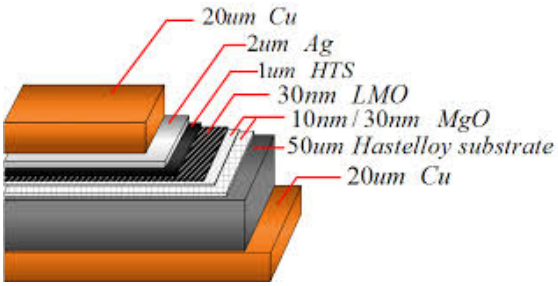


Figura 2.4: Estructura interna de una cinta HTS. [14]

2.2.3. Entorno de Pruebas & Definición del Problema

A continuación, se procederá a describir las propiedades de la cinta superconductora, la geometría del sistema SMES y el transitorio eléctrico empleados para simular y validar el cálculo de pérdidas AC.

Propiedades de la cinta superconductora

La cinta superconductora se compone de múltiples capas de distintos materiales, donde solo la capa en la mitad es superconductora. Para el cálculo de pérdidas AC solo se tendrá en cuenta la naturaleza de la capa superconductora, pues en la práctica es por donde circula mayoritariamente la intensidad, aunque no se ha de olvidar que en la realidad se inducen corrientes entre las diferentes capas durante los transitorios eléctricos debido al sometimiento a un campo magnético variable y a sus resistividades no nulas. Sin embargo, debido a la poca significancia de estos efectos, el resto de materiales se tratarán con las propiedades electromagnéticas del aire. La descomposición de los distintos materiales de la estructura interna de una cinta superconductora se muestra en la Figura 2.4.

La cinta en particular de estudio presenta las características geométricas y superconductoras especificadas en la tabla 2.1. Para la densidad de corriente crítica se va a obviar su dependencia con la temperatura, fijando esta durante todo el problema a 4.2 K. Esta consideración se sustenta sobre la asunción de que el sistema de refrigeración es lo suficientemente potente como para no permitir incrementos de temperatura considerables durante los transitorios eléctricos. Además, considerar la temperatura como una variable, supone la resolución de un problema de transmisión de calor completamente acoplado con el problema electromagnético, elevando considerablemente la complejidad del problema, consecuentemente ralentizadno su resolución.

Propiedades cinta HTS	**Valor**
Altura de la cinta (h)	4.8 mm
Espesor capa superconductora (w)	33 μm
Espesor total cinta (Sep)	340μm
Propiedades capa superconductora	**Valor**
Exponente del superconductor (n)	40
Campo eléctrico crítico (E_c)	0,1 mV/m
Densidad de corriente crítica (J_c)	$f(\|\|B\|\|, \theta)$ (A/m^2)

Cuadro 2.1: Propiedades de la cinta HTS y de la capa superconductora.

Para la dependencia de J_c con $\|B\|$ y θ se dispone de una base de datos calculada experimentalmente para la cinta superconductora del fabricante que posee Cyclomed Technologies. Asimismo, la siguiente expresión analítica ha sido calculada y comprobada empíricamente para relacionar J_c con el campo magnético y el ángulo de incidencia para una determinada cinta a 4.2 K [11]:

$$I_c = \frac{k_0}{(|B| + \beta_0)^{\alpha_0}} + \frac{k_1}{(|B| + \beta_1)^{\alpha_1}} \cdot \left[(\omega_1^2 \cdot \cos^2(\theta - \varphi_1) + \sin^2(\theta - \varphi_1))\right]^{-1/2} \quad (2.31)$$

donde I_c es la intensidad crítica de la cinta superconductora y ω_1 una función del campo magnético, definida como:

$$\omega_1 = c \left[|B| + \left(\frac{1}{c}\right)^{5/3} \right] \quad (2.32)$$

Para calcular la densidad de corriente crítica, la intensidad crítica debe dividirse por la sección de la cinta. Los parámetros k_0, k_1, β_0, β_1, α_1, ϕ_1 y c se definen para ajustar la expresión analítica a los valores experimentales de acuerdo con el estudio por D K Hilton, A V Gavrilin y U P Trociewitz [11]. No obstante, algunos de estos parámetros se han modificado para ajustar la expresión analítica a la intensidad crítica correspondiente a la cinta de estudio.

Propiedades de la geometría

Para comprobar el comportamiento del modelo se han escogido tres geometrías distintas: una primera geometría formada por dos bobinas conectadas en serie una encima de otra, en formato doble pancake, constituidas por 20 espiras de cinta superconductora, que a lo largo de la tesis será referida como (20,2); la segunda geometría consta de dos bobinas en formato doble pancake formadas por 40 vueltas de cinta superconductora y será referida como (40,4); la última

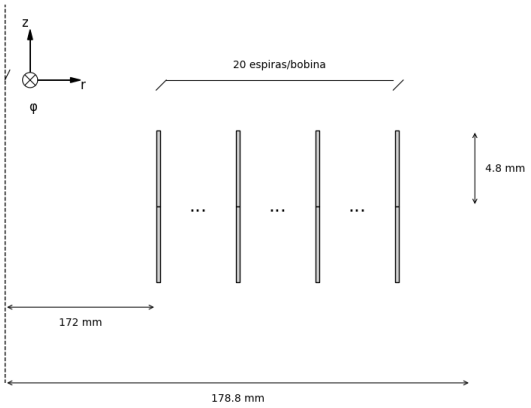


Figura 2.5: Geometría (20,2) del Sistema SMES.

geometría se compone de un doble pancake con 144 vueltas por pancake (144,2). La representación 2.5 muestra la primera geometría del sistema SMES a simular sin escala, donde la línea punteada indica el eje de simetría de revolución. Todos los parámetros se mantienen para la segunda y tercera geometría excepto el número de bobinas, el número de espiras y el radio externo, que depende del número de vueltas de superconductor. Cabe puntualizar que al estar las bobinas conectadas en serie la intensidad que circula a través de cualquier sección transversal de la cinta siempre será la misma para todo instante de tiempo. Se han escogido tres geometrías arbitrarias para poder visualizar diferentes rendimientos de los modelos para complejidades incrementales del problema.

Propiedades del transitorio

Como problema a simular se ha optado por un transitorio de 10 s compuesto por una sinusoide con una frecuencia de 1 Hz y un valor de pico de 600 A. Se ha escogido esta evolución de la intensidad porque se considera que es lo suficientemente compleja y realista como para estudiar la evolución de las pérdidas AC. Los siguientes capítulos presentan diversos modelos que simularán este transitorio, junto con las propiedades geométricas y superconductoras especificadas para predecir las pérdidas AC.

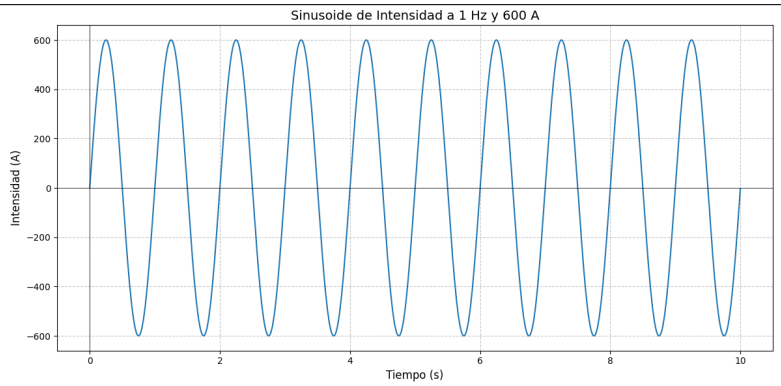


Figura 2.6: Transitorio eléctrico a simular.

Hardware

Todos los modelos desarrollados en Julia, así como el modelo en Matlab se resolverán con un ordenador cuyas características son:

- Procesador: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz 2.81 GHz
- RAM instalada: 12,0 GB
- Tipo de sistema: Sistema operativo de 64 bits, procesador basado en x64
- Disco SSD del sistema: 932,0 GB
- Tarjeta gráfica: NVIDIA GeForce GTX 1050 (2 GB), Intel(R) HD Graphics 630 (128 MB)

2.2.4. Evaluación de Resultados

El Cuadro 2.2 presenta un desglose detallado de los tiempos de ejecución asociados a cada una de las funciones implicadas en la simulación de geometrías de bobinas para un sistema SMES, junto con el parámetro Q, que mide la potencia disipada por el sistema y será particularmente interesante como medida de la precisión de los modelos de Julia. Para evaluar el impacto de la complejidad geométrica del sistema en la eficiencia computacional en términos de tiempo del modelo se estudian dos configuraciones distintas $(20, 2)$, $(40, 4)$ y $(144, 2)$. La notación (N, M) denota una configuración de M bobinas, cada una de las cuales consta de N espiras; así, por ejemplo, la configuración $(20, 2)$ corresponde a un sistema de dos bobinas con 20 espiras cada una. Cabe destacar que las configuraciones geométricas (N, M) se eligieron sin un criterio específico más allá del incremento de la complejidad total de la geometrías. Esto se debe a que, al considerar el

producto N×M en la resolución del problema el impacto en la escala es lineal, haciendo que un aumento en N sea equivalente a un aumento proporcional en M.

Un análisis preliminar de los datos revela una evidente correlación entre el incremento en la complejidad geométrica y el aumento de los tiempos de procesamiento. Si bien ciertas funciones, tales como `setMeshSuperconductor2PancakeCoils`, `BuildGeometry12PancakeCoils` y `ApplySymmetry12PancakeCoils`, exhiben un incremento en sus tiempos de ejecución que, en términos absolutos, puede considerarse despreciable en el contexto global del rendimiento computacional, otras rutinas demuestran una dependencia notablemente más pronunciada.

Específicamente, la función `CreationDphiMatrix2PancakeCoils` manifiesta un incremento más notorio en su tiempo de ejecución al progresar hacia geometrías más elaboradas, sugiriendo una sensibilidad considerable a la escala del problema, si bien su contribución es marginal en términos absolutos de eficiencia. Sin embargo, la identificación de *cuellos de botella* computacionales se localiza inequívocamente en las funciones `CreationVectorPotentialMatrix12PancakeCoils` y `CreationMagneticFieldMatrix12PancakeCoils`. Para ilustrar esta dependencia, al transitar de la geometría $(20, 2)$ a la $(40, 4)$, lo que implica un incremento de cuatro veces en los parámetros característicos de la geometría, se observa una amplificación de los tiempos de ejecución que se aproxima a un factor de diez para ambas funciones.Asimismo, la transición de la configuración (40, 4) a la (144, 2), que representa un factor de incremento de aproximadamente dos en los parámetros geométricos, desencadena un aumento de los tiempos de ejecución que excede un factor de tres.

En resumen, los resultados empíricos contenidos en el Cuadro 2.2 demuestran que, a medida que la complejidad inherente a la geometría del problema incrementa, los tiempos de ejecución de las simulaciones experimentan un aumento no lineal, evidenciando una clara problemática de escalabilidad. Esta observación **pone de manifiesto y cuantifica las limitaciones en la eficiencia computacional que han sido previamente discutidas**, representando un gran obstáculo para la operatividad y la eficiencia del modelo. No solo se compromete el rendimiento, sino que también surgen restricciones fundamentales para realizar simulaciones en regímenes de cuasi-tiempo real con baja latencia para el desarrollo y la implementación de aplicaciones avanzadas en campos que dependen de la modelización precisa de fenómenos electromagnéticos. La optimización de estas funciones, particularmente `CreationVectorPotentialMatrix12PancakeCoils` y `CreationMagneticFieldMatrix12PancakeCoils`, se erige como una prioridad fundamental para mitigar las limitaciones de rendimiento y posibilitar la viabilidad de escenarios de simulación de mayor envergadura.

Cuadro 2.2: Tiempos de Ejecución para Diferentes Geometrías en MATLAB

Geometría (NxM)	**(20,2)**	**(40,4)**	**(144,2)**
setMeshSuperconductor2PancakeCoils [seg]	0.0184	0.0882	0.1418
BuildGeometry12PancakeCoils [seg]	0.0020	0.0024	0.0030
CreationVectorPotentialMatrix12PancakeCoils [seg]	258.51	2193.1	6863.9
CreationMagneticFieldMatrix12PancakeCoils [seg]	667.83	4814.4	15334
CreationDphiMatrix2PancakeCoils [seg]	0.0775	6.5275	48.9914
ApplySymmetry12PancakeCoils [seg]	0.0353	0.2025	0.9235
ODE Solver [seg]	400.92	4923.2	27425
Q [W]	186.5815	2171.0	12289

Análisis de la Dinámica de Potencia, Densidad de Corriente y Generación de Calor

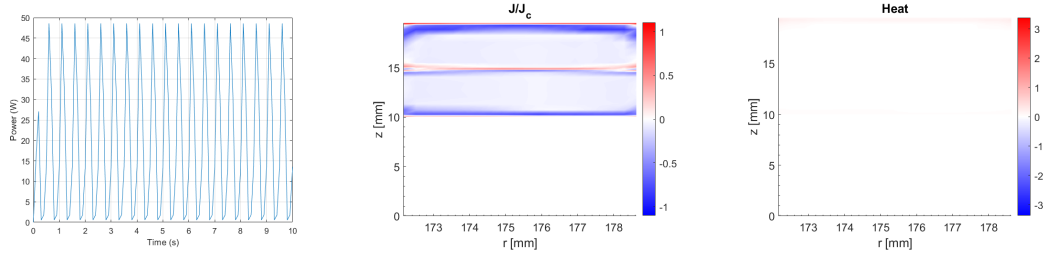
Las Figuras 2.7, 2.8 y 2.9 ofrecen una representación visual de la dinámica de la potencia disipada, de la distribución normalizada de la densidad de corriente (J/J_c) y de la generación de calor en función de la geometría de las bobinas para los escenarios $(20, 2)$, $(40, 4)$ y $(144, 2)$ respectivamente. Estas visualizaciones permiten entender el comportamiento termodinámico y electromagnético de un sistema SMES bajo las condiciones operativas simuladas.

En la figura central en 2.7 a 2.9 ilustra la distribución de la densidad de corriente normalizada con la densidad de corriente crítica (J/J_c) en las bobinas. Se observa consistentemente que la densidad de corriente tiende a concentrarse en los bordes externos de las espiras, mientras que su valor se aproxima a cero en las regiones internas. Este comportamiento es una manifestación del efecto *skin* o de las no-linealidades inherentes a los superconductores de alta temperatura crítica.

Por otra parte, la figura izquierda presenta la curva de potencia disipada en el tiempo. Esta curva refleja las pérdidas energéticas del sistema y su comportamiento cíclico es indicativo de la respuesta del superconductor a una excitación periódica, lo cual concuerda con la naturalez sinoidal de la corriente de ajuste I_{set}. El valor de `Q` en el Cuadro 2.2, que representa la integral de esta curva de potencia a lo largo del tiempo, cuantifica la energía total disipada y corrobora el aumento significativo de las pérdidas energéticas con el incremento de la complejidad geométrica.

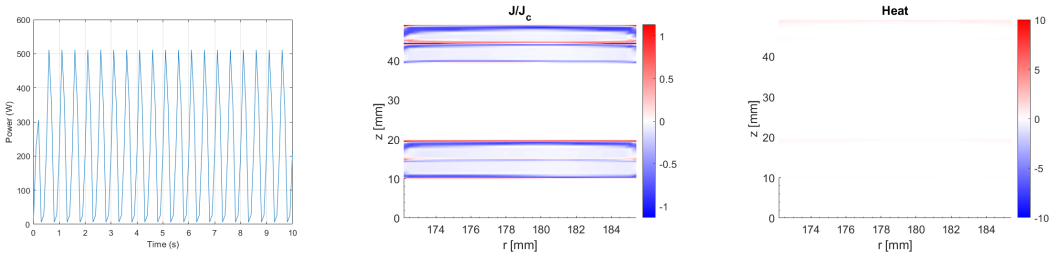
Finalmente, la figura situada a la derecha muestra la distribución espacial del calor generado en las bobinas. El conocimiento de la distribución de la densidad de corriente es fundamental para determinar la distribución de potencia generada y consecuentemente las áreas donde la bobina experimentará un mayor calentamiento. En correlación con la distribución de corriente, el calor se genera predominantemente en los bordes externos de las espiras debido a la concentración de

la densidad de corriente en esas regiones. De acuerdo con esta explicación y como muestran las Figuras 2.7 a 2.9 para el transitorio simulado el calentamiento se concentra principalemente en el extremo superior de los *pancakes*, pues es la zona con mayor concentración de densidad de corriente. Estos patrones de calentamiento son críticos para el diseño térmico y la gestión de la estabilidad de las bobinas superconductoras.



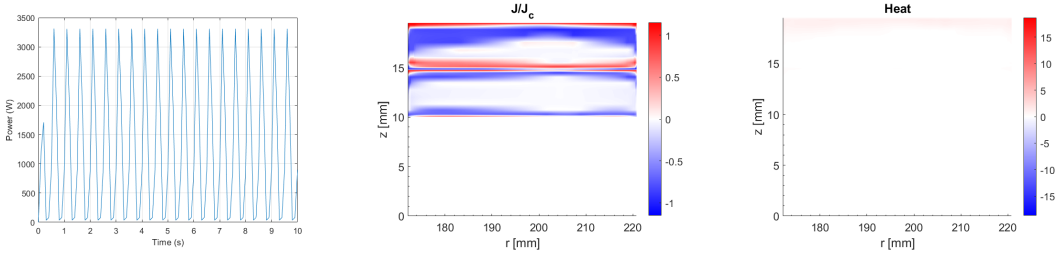
(a) Potencia [W] (b) J/J_c (c) Calor

Figura 2.7: Distribuciones en MATLAB para la Geometría (20,2).



(a) Potencia [W] (b) J/J_c (c) Calor

Figura 2.8: Distribuciones en MATLAB para la Geometría (40,4).



(a) Potencia [W] (b) J/J_c (c) Calor

Figura 2.9: Distribuciones en MATLAB para la Geometría (144,2).

Dadas las limitaciones de eficiencia computacional observadas, particularmente el incremento desproporcionado en los tiempos de ejecución conforme aumenta la complejidad geométrica del sistema, la implementación de técnicas avanzadas de modelado computacional, así como la migración al lenguaje Julia se postulan como soluciones de interés para alcanzar un modelo más eficiente. Aunque se profundizará en la siguiente sección, la elección el lenguaje Julia se debe a su concepción como un lenguaje de programación que combina la interactividad de entornos como MATLAB o Python con la productividad para el trabajo técnico de MATLAB, y la velocidad de C. De esta manera se abre la puerta a una iniciativa fundamental: migrar y optimizar el modelo de MATLAB para permitir la realización de simulaciones con una mayor resolución espacial y temporal en tiempos computacionalmente más razonables.

Capítulo 3

Desarrollo de nuevos Modelos

Todos los modelos desarrollados se basan en los métodos de cálculo implementados en el modelo de MATLAB de J. Orea [7], que se han expuesto en el capítulo anterior. En este capítulo se expondrán secuencialmente los modelos desarrollados en Julia , explicando las mejoras de software y de hardware incluidas de manera incremental para alcanzar el modelo definitivo que presenta un notable incremento en la eficiencia sin detrimento en la precisión del cálculo de pérdidas AC.

Aunque las limitaciones mencionadas en el apartado anterior 2.2.2 están principalmente ligadas a ineficiencias en la configuración e implementación del modelo, Julia es un lenguaje de programación diseñado específicamente para la computación técnica y científica de alto rendimiento, cuyas características lo convierten en una alternativa más atractiva y con mayor potencial para aplicaciones computacionalmente intensas frente a MATLAB [15]. Las siguientes son algunas de las principales razones por las que se seleccionó Julia como lenguaje objetivo para desarollar las mejoras del modelo base:

- **Diseñado para la Computación Científica:** Julia fue creado considerando las necesidades de la comunidad científica [16]. Ofrece tipos numéricos eficientes, soporte nativo para operaciones con matrices, un sistema de tipos flexible y un gestor de paquetes robusto que incluye bibliotecas de alto rendimiento para álgebra lineal, optimización, y, computación paralela tanto en CPU como en GPU. Además, este lenguaje resuelve inherentemente el "problema de los dos lenguajes" (*two-language problem*) [15], un desafío frecuente en el desarrollo de software científico, donde los desarrolladores necesitan usar dos lenguajes distintos. Un primer lenguaje de alto nivel como MATLAB o R para realizar el prototipado inicial mediante distintos ensayos en un lenguaje flexible, pero más lento. Este prototipado es seguido por el traslado del código a un lenguaje de bajo nivel como C++ o Fortran con menor flexibilidad pero más eficiente para realizar la implementación completa.

- **Velocidad Comparable a C/Fortran con Sintaxis de Alto Nivel:** Se trata de un lenguaje compilado *Just-In-Time* (JIT) que genera código máquina optimizado en tiempo de ejecución, alcanzando un rendimiento comparable a lenguajes compilados de bajo nivel como C o Fortran, mientras mantiene una sintaxis de alto nivel similar a MATLAB o Python, que facilita la legibilidad, la escritura de algoritmos [15]. Mediante esta combinación se logra hacer frente al mencionado "problema de los dos lenguajes" exitosamente. Como evidencia la Figura 3.1, aunque MATLAB es muy rápido, la eficiencia de Julia es casi inigualable por el resto de lenguajes.

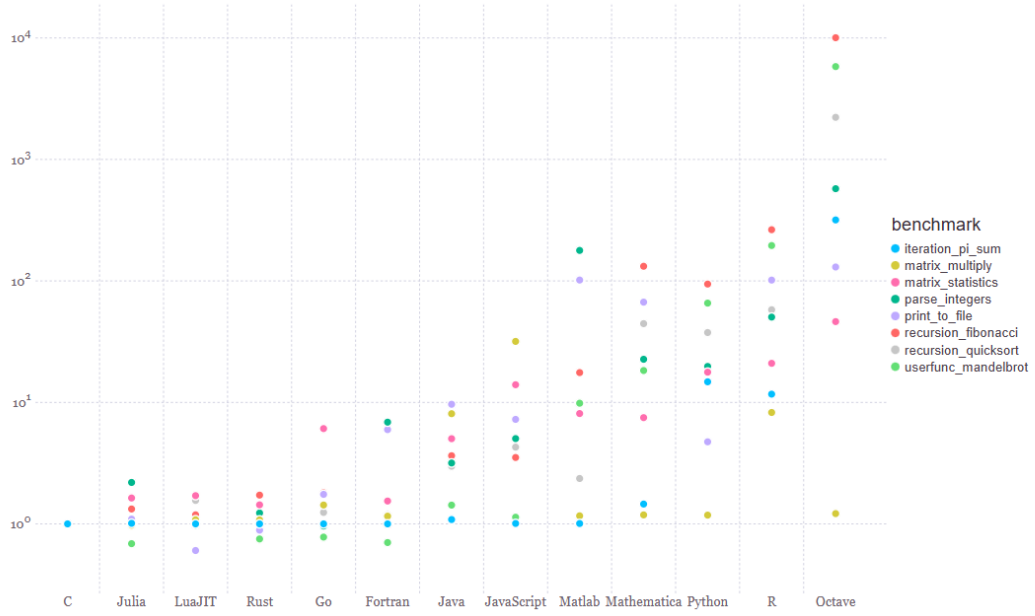


Figura 3.1: Benchmark Times Normalized against C Implementation (Source: [17])

- **Compilación JIT:** Las funciones se compilan la primera vez que se ejecutan con un conjunto particular de tipos de argumentos, aplicando un amplio conjunto de optimizaciones, como simplificación y eliminación de código muerto, optimizaciones de bucles, o vectorización, y generando el código máquina específico para la plataforma [18]. Este proceso inicial y exhaustivo suele implicar que la primera ejecución sea ligeramente más lenta. Sin embargo, una vez que el código máquina ha sido generado y enlazado, las compilaciones posteriores para el mismo conjunto de tipos de argumentos son muy rápidas, puesto que el código ya está optimizado y listo para ser ejecutado.
- **Metaprogramación y Generación de Código:** La metaprogramación es la capacidad del código para manipular y generar otro código. Esta carac-

terística avanzada es clave para crear optimizaciones muy sofisticadas que producen código altamente eficiente, como las usadas en librerías de paralelismo. A diferencia de otros lenguajes Julia aborda dos de los grandes desafíos críticos en el ámbito científico: el "problema de los dos lenguajesz el "problema de la expresión", facilitando la inclusión de nuevos tipos de datos y operaciones en código existente sin incurrir en problemas de compatibilidad [15]. Así, la metaprogramación no solo potencia el rendimiento de Julia, sino que también agiliza la colaboración y el desarrollo del software.

- **Paralelismo Integrado y Ecosistema GPU:** La capacidad de Julia para el procesamiento paralelo es una ventaja clave, pues reduce significativamente el tiempo de ejecución de computaciones complejas. Su soporte nativo para el paralelismo abarca tanto la ejecución en CPU, a través de *multithreading* y computación distribuida, como en GPU. La computación en GPU, particularmente relevante para este proyecto, goza de un ecosistema maduro proporcionado por el paquete CUDA.jl [19]. Este paquete permite la escritura directa de kernels CUDA en Julia, eliminando la necesidad de lenguajes como C++ y simplificando notablemente la integración de la aceleración por hardware en el código existente[20].

3.1. Modelo I: Migración de Matlab a Julia & Vectorización

3.1.1. Proceso de Migración y Vectorización del Código

El proceso de portar el código de MATLAB a Julia siguió una estrategia incremental, centrándose incialmente en una correcta implementación de la funcionalidad y posteriormente en la optimización para disminuir el tiempo de ejecución. El primer modelo se basa en una **traducción literal de las funciones y algoritmos matemáticos** de MATLAB a Julia, fase facilitada por la similitud existente entre ambos lenguajes de programación. Durante toda esta primera fase se fue comprobando reiterada y exhaustivamente la **fidelidad del nuevo código**. Las salidas del modelo en Julia fueron comparadas con el modelo original en MATLAB, verificando así la equivalencia entre ambos.

Concluida y verificada la migración del código al nuevo entorno, se procedió a la **vectorización** del mismo. El código en MATLAB no solo presenta un exceso de bucles for, sino una falta de optimización, pues muchas operaciones se implementan en bucles individuales en lugar de fusionarse. Para solventar esta ineficiencia y aprovechar las operaciones vectorizadas en Julia en la CPU, se llevó a cabo la primera etapa de optimización del rendimiento. La vectorización es una técnica que

consiste en aplicar operaciones sobre conjuntos de datos completos, como vectores o matrices, en lugar de iterar por cada uno de los elementos por medio de bucles explícitos.

Aunque la vectorización internamente lleva a cabo un bucle, los bucles explícitos incurren en una **sobrecarga por cada iteración**, mientras que esta sobrecarga no se produce en las operaciones vectorizadas. Estas realizan una única entrega a una función de biblioteca optimizada, generalmente implementada en C o Fortran, como BLAS, que operan sobre el conjunto completo. Además, las operaciones vectorizadas permiten al compilador generar instrucciones ***Single Instruction, Multiple Data*** **(SIMD)**, que realizan simultáneamente la misma operación en múltiples elementos de datos haciendo uso de las unidades de procesamiento vectoriales de la CPU. La eficiencia se beneficia tanto del paralelismo de la CPU como de un acceso más eficiente a memoria. Al operar sobre bloques contiguos de memoria, las operaciones vectorizadas y funciones optimizadas para matrices y vectores tienden a tener una **mejor localización en caché**, reduciendo los costes de acceso a la memoria principal. Asimismo, las operaciones vectorizadas **asignan y rellenan grandes bloques de memoria más eficientemente**, al trabajar con bloques de datos en lugar de elementos individuales, lo cual reduce el número de operaciones a nivel de CPU.

Los Algoritmos 1 y 2 muestran la implementación de la función `CreationDphi Matrix2PancakeCoils` en MATLAB y en su respectiva versión optimizada en Julia. Esta función construye la matriz de superficie S para la geometría de una bobina de doble pancake. Dicha matriz relaciona la distribución de corriente en la bobina con la restricción de corriente total. S se emplea en el sistema de EDOs para que la suma de densidades de corriente en cada espira corresponda a la corriente de transporte impuesta $Iset$, restricción necesaria para modelar el comportamiento electromagnético de la bobina under transport currentbajo corriente de transporte. La versión inicial en MATLAB hace uso de bucles anidados para asignar valores a la matriz `S` en cada espira, iterando fila por fila dentro de cada bloque.

La versión migrada a Julia refactoriza este proceso, reemplazando el bucle anidado por una asignación de bloque directa. La función `repeat` y la transposición permiten que la compilación JIT de Julia realice una operación de asignación de bloque de la matriz `S` mucho más eficiente en un solo tiempo. Si bien este cambio es representativo de las pequeñas optimizaciones aplicadas para evitar bucles explícitos y realizar operaciones de matriz sobre bloques, el elemento diferencial se encuentra en la computación de las matrices de densidad de corriente `K` (2.26) y del campo magnético `M` (2.29), que suponen los mayores cuellos de botella.

Algorithm 1 Creation of Surface Matrix for Pancake Coils

Function: CreationDphiMatrix2PancakeCoils(*geometry*,*coil*)
Input:
  geometry: A structure containing the following fields:
    s: A 1D array representing surface elements.
    Nz: Number of divisions in the z-direction (axial).
    Nr: Number of divisions in the r-direction (radial).
    Nc: Number of coils in the system.
    Ne: Number of turns (espira) per coil.
Output:
  S: Matrix representing surface elements for double pancake coil config.
$N \leftarrow Nz \cdot Nr \cdot Nc \cdot Ne$
$S \leftarrow N \times N$ matrix of zeros
for $N\_espira = 1$ to $(2 \cdot Ne \cdot Nc)$ **do**
  ▷ Iterate through each turn in the double pancake coil
  $start\_idx \leftarrow (N\_espira - 1) \cdot Nr \cdot Nz + 1$
  $end\_idx \leftarrow N\_espira \cdot Nr \cdot Nz$
    for $i = start\_idx$ to end_idx **do**
    ▷ Assign relevant section of s array defined by *pos* to the current
      $S[i, pos] \leftarrow s[pos]$
return S

Algorithm 2 Optimized Creation of Surface Matrix for Pancake Coils

Function: CreationDphiMatrix2PancakeCoils(*geometry*,*coil*)
Input:
  geometry: A structure/object containing the following fields:
    s: A 1D array representing surface elements.
    Nz: Number of divisions in the z-direction (axial).
    Nr: Number of divisions in the r-direction (radial).
    Nc: Number of coils in the system.
    Ne: Number of turns (espira) per coil.
  coil: A structure/object that will store the resulting surface matrix.
Output:
  S: A matrix representing the surface elements, optimized for pancake coil.
$N \leftarrow Nz \cdot Nr \cdot Nc \cdot Ne$
$S \leftarrow N \times N$ matrix of zeros
for $N\_espira = 1$ to $(Ne \cdot Nc)$ **do**
    ▷ This loop operates at block level.
  $start\_pos \leftarrow (N\_espira - 1) \cdot Nr \cdot Nz + 1$
  $end\_pos \leftarrow N\_espira \cdot Nr \cdot Nz$
  $pos \leftarrow$ range from $start\_pos$ to end_pos
  $row\_range \leftarrow (N\_espira - 1) \cdot Nz \cdot Nr + 1 : (N\_espira) \cdot Nz \cdot Nr$
  $S[row\_range, pos] \leftarrow \text{repeat}(s[pos], 1, Nz \cdot Nr)'$
$coil.Ss\_super \leftarrow S$
return S

Cálculo de las Matrices de Vector Potencial Eléctrico K y de Campo Magnético M

Las funciones `CreationVectorPotentialMatrix12PancakeCoils` 4 y `CreationMagneticFieldMatrix12PancakeCoils` 3 son las implementaciones en software directas de las formulaciones teóricas presentadas en las ecuaciones 2.26 y 2.29 respectivamente. Las diferencias en la implementación en `MATLAB` y en `Julia` de la creación de las matrices radican principalmente en la adopción de buenas prácticas de programación, la eficiencia computacional, la claridad y la concisión del código para el cómputo de las integrales numéricas. Originalmente el cálculo de las integrales K y M en `MATLAB` se encuentra anidado dentro de dos bucles análogos, que si bien representan una aproximación directa de la formulación matemática, supone una gran carga computacional. Además, a pesar de compartir las estructuras de bucle, las matrices están segregadas en dos funciones distintas, 4 y 3 respectivamente. En el enfoque en `Julia`, sin embargo, se ha optado por la unificación de las funciones en una única como se muestra en el Algoritmo 5, consolidando la lógica de cálculo para ambas matrices en un único bucle principal. La supresión del bucle anidado se deriva de la vectorización de las operaciones en las integrales, donde las operaciones aritméticas escalares básicas $(+, -, *, /)$, así como las funciones matemáticas elementales como el logaritmo, la exponencial y la raíz cuadrada, son sustituidas por sus equivalentes de "dot operation" $(.+, .-, .*, ./, \log.(), \exp.(), \sqrt{.})$. Al aplicar las operaciones elemento a elemento sobre los vectores y matrices se logra una mayor concisión y optimización del código con un único bucle para la iteración principal.

Algorithm 3 Creation of Magnetic Field Matrix for 2 Pancake Coils

Function: CreationMagneticFieldMatrix12PancakeCoils(*geometry*)
Input:
  geometry: A structure containing geometric parameters, including:
    r_middle: Radial midpoints for each coil segment.
    z_middle: Axial midpoints for each coil segment.
    r_low: Lower radial bounds for integration.
    r_high: Upper radial bounds for integration.
    z_low: Lower axial bounds for integration.
    z_high: Upper axial bounds for integration.
Output:
  K_{Br}: Matrix representing the radial component of the magnetic field.
  K_{Bz}: Matrix representing the axial component of the magnetic field.
$N \leftarrow$ length($geometry.r\_middle$)
$K_{Br} \leftarrow N \times N$ matrix of zeros
$K_{Bz} \leftarrow N \times N$ matrix of zeros

for $i = 1$ to N **do**
  $ri \leftarrow geometry.r\_middle(i)$
  $zi \leftarrow geometry.z\_middle(i)$
  for $j = 1$ to N **do**
    $aj \leftarrow geometry.r\_low(j)$
    $bj \leftarrow geometry.r\_high(j)$
    $cj \leftarrow geometry.z\_low(j)$
    $dj \leftarrow geometry.z\_high(j)$
    $ej \leftarrow (aj + bj)/2$
    if $|i - j| < 4$ **then**
      if $i = j$ **then**
        $K_{Bz}(i, j) \leftarrow 0$
        $K_{Br}(i, j) \leftarrow 0$
      else
        Define $integrandz(r_{ac}, \theta_{ac})$ as:

$$\frac{r_{ac}(r_{ac} - ri\sin(\theta_{ac}))(dj - zi)}{(r_{ac}^2 - 2\sin(\theta_{ac})r_{ac}ri + ri^2)((dj - zi)^2 + r_{ac}^2 + ri^2 - 2r_{ac}ri\sin(\theta_{ac}))^{1/2}} - (\dots(\text{with } cj \text{ instead of } dj)\dots)$$

        Define $integrandr(r_{ac}, \theta_{ac})$ as:

$$\frac{r_{ac}\sin(\theta_{ac})}{((dj - zi)^2 + r_{ac}^2 + ri^2 - 2r_{ac}ri\sin(\theta_{ac}))^{1/2}} - (\dots(\text{with } cj \text{ instead of } dj)\dots)$$

        $K_{Bz}(i, j) \leftarrow \text{integral2}(integrandz, aj, bj, -\pi/2, \pi/2, \text{RelTol}, 1e-6, \text{AbsTol}, 1e-9)$
        $K_{Br}(i, j) \leftarrow \text{integral2}(integrandr, aj, bj, -\pi/2, \pi/2, \text{RelTol}, 1e-6, \text{AbsTol}, 1e-9)$

34:

35: **else**

▷ Far-field approximation

36: Define $integrandz(\theta_{ac})$ as:

$$\frac{bj-aj}{6}\left(\left(\frac{aj(aj-ri\sin(\theta_{ac}))(dj-zi)}{(aj^2-2\sin(\theta_{ac})ajri+ri^2)((dj-zi)^2+aj^2+ri^2-2ajri\sin(\theta_{ac}))^{1/2}}\right.\right.$$
$$-\ldots(\text{with } cj \text{ instead of } dj)\ldots)$$
$$+4(\ldots(\text{with } ej \text{ instead of } aj)\ldots)$$
$$\left.\left.+(\ldots(\text{with } bj \text{ instead of } aj)\ldots)\right)\right.$$

37: Define $integrandr(\theta_{ac})$ as:

$$\frac{bj-aj}{6}\left(\left(\frac{aj\sin(\theta_{ac})}{((dj-zi)^2+aj^2+ri^2-2ajri\sin(\theta_{ac}))^{1/2}}\right.\right.$$
$$-\ldots(\text{with } cj \text{ instead of } dj)\ldots)$$
$$+4(\ldots(\text{with } ej \text{ instead of } aj)\ldots)$$
$$\left.\left.+(\ldots(\text{with } bj \text{ instead of } aj)\ldots)\right)\right.$$

38: $K_{Bz}(i,j) \leftarrow \text{integral}(integrandz, -\pi/2, \pi/2, \text{RelTol}, 1e-6, \text{AbsTol}, 1e-9)$

39: $K_{Br}(i,j) \leftarrow \text{integral}(integrandr, -\pi/2, \pi/2, \text{RelTol}, 1e-6, \text{AbsTol}, 1e-9)$

40: $K_{Bz} \leftarrow \text{real}(K_{Bz}/10^3)$

41: $K_{Br} \leftarrow \text{real}(K_{Br}/10^3)$

42: $K_{Bz}(\text{isNaN}(K_{Bz})) \leftarrow 0$

43: $K_{Br}(\text{isNaN}(K_{Br})) \leftarrow 0$

44: **return** K_{Br}, K_{Bz}

Existen dos cambios, que aunque de menor calibre, mejoran la legibilidad y la modularidad del código. Por un lado la función `CreationMagneticFieldMatrix12PancakeCoils` designa a sus salidas como K. A lo largo de la tesis se ha identificado el ámbito de la magnetoestática con la letra M. Si bien este cambio no implica una mejora en la computación del código, contribuye a la legibilidad e interpretabilidad del modelo, para evitar confusión o inducciones a error.

Algorithm 4 Creation of Vector Potential Matrix for 2 Pancake Coils

Function: CreationVectorPotentialMatrix12PancakeCoils(*geometry*)
Input:
  geometry: A structure containing geometric parameters, including:
    r_middle: Radial positions for the center of elements.
    z_middle: Axial positions for the center of elements.
    r_low: Lower radial bounds for integration.
    r_high: Upper radial bounds for integration.
    z_low: Lower axial bounds for integration.
    z_high: Upper axial bounds for integration.
Output:
  K: The calculated Vector Potential Matrix (in units of m^2).
$N \leftarrow \text{length}(geometry.r\_middle)$
$K \leftarrow N \times N$ matrix of zeros
for $i = 1$ to N **do**
  $ri \leftarrow geometry.r\_middle(i)$
  $zi \leftarrow geometry.z\_middle(i)$
  for $j = 1$ to N **do**
    $aj \leftarrow geometry.r\_low(j)$
    $bj \leftarrow geometry.r\_high(j)$
    $cj \leftarrow geometry.z\_low(j)$
    $dj \leftarrow geometry.z\_high(j)$
    $ej \leftarrow (aj + bj)/2$
    if $(i \neq j)$ **then**
      ▷ Mutual interaction term calculation
      Define $integrand_{mut}(\theta_{ac})$ as:
      $\frac{bj-aj}{6} \cdot \left(\log \left(\frac{\sqrt{(zi-cj)^2+ri^2+aj^2-2\cdot ri\cdot aj\cdot\cos(\theta_{ac})}+zi-cj}{\sqrt{(zi-dj)^2+ri^2+aj^2-2\cdot ri\cdot aj\cdot\cos(\theta_{ac})}+zi-dj} \right) \cdot aj \cdot \cos(\theta_{ac}) \right.$
      $+4 \cdot \log(\dots \text{ (with } ej \text{ instead of } aj) \dots) \cdot ej \cdot \cos(\theta_{ac})$
      $\left. + \log(\dots \text{ (with } bj \text{ instead of } aj) \dots) \cdot bj \cdot \cos(\theta_{ac}) \right)$
      $K[i,j] \leftarrow \text{integral}(integrand_{mut}, 0, \pi, \text{RelTol}, 1e-6, \text{AbsTol}, 1e-9)$
    else
      ▷ Self-interaction term calculation
      Define $integrand_{self}(r_{ac}, \theta_{ac})$ as:
      $\log \left(\frac{\sqrt{(zi-cj)^2+ri^2+r_{ac}^2-2\cdot ri\cdot r_{ac}\cdot\cos(\theta_{ac})}+zi-cj}{\sqrt{(zi-dj)^2+ri^2+r_{ac}^2-2\cdot ri\cdot r_{ac}\cdot\cos(\theta_{ac})}+zi-dj} \right) \cdot r_{ac} \cdot \cos(\theta_{ac})$
      $K[i,j] \leftarrow \text{integral2}(integrand_{self}, aj, bj, 0, \pi, \text{RelTol}, 1e-6, \text{AbsTol}, 1e-9, \text{Method}, \text{auto})$
return $real(K/10^6)$

El segundo cambio respecta al cálculo de las integrales en `MATLAB`, caracterizado por una definición interna. En `Julia` las integrales se definen de manera externa en funciones propias, incrementando la reusabilidad de las mismas y simplificando la estructura del bucle principal. Estos dos cambios, aunque menos trascendentes para el propósito de esta tesis contribuyen a un mejor mantenimiento y estructuración del modelo.

Por último, el ecosistema de paquetes de `Julia` diverge del de `MATLAB`, por ello las funciones `integral` e `integral2` se sustituyen en 5 por `quadgk` y `hcubature` respectivamente. El paquete `quadgk` es la elección predominante para la integración unidimensional adaptativa en `Julia`. Su algoritmo se ajusta dinámicamente a la función para lograr la precisión deseada, definida por la tolerancia relativa (`rtol`) y absoluta (`atol`). Este enfoque adaptativo asegura una **alta precisión** con un número eficiente de evaluaciones de la función. `Hcubature` está diseñado específicamente para la integración multidimensional, aunque algunas de sus implementaciones emplea internamente `quadgk`. La elección entre ambas alternativas en función de la dimensionalidad del problema permite abordar en Julia los problemas de integración numérica con gran robustez y eficiencia.

Algorithm 5 Magnetic Field and Vector Potential Matrix Calculations Julia CPU

Function: CreationPotentialMagneticMatrix12PancakeCoils(*geometry*)
Input:
  geometry: A structure containing geometric parameters, including:
    r_middle: Radial midpoints for each coil segment.
    z_middle: Axial midpoints for each coil segment.
    r_low: Lower radial bounds for integration (aj).
    r_high: Upper radial bounds for integration (bj).
    z_low: Lower axial bounds for integration (cj).
    z_high: Upper axial bounds for integration (dj).
Output:
  K: Matrix representing the vector potential.
  MBr: Matrix representing the radial component of the magnetic field.
  MBz: Matrix representing the axial component of the magnetic field.
$N \leftarrow$ length($geometry.r\_middle$)
$K \leftarrow$ Initialize $N \times N$ matrix of Float64
$MBr \leftarrow$ Initialize $N \times N$ matrix of Float64
$MBz \leftarrow$ Initialize $N \times N$ matrix of Float64
$temp\_K \leftarrow$ Initialize N-element vector of Float64
$temp\_MBz \leftarrow$ Initialize N-element vector of Float64
$temp\_MBr \leftarrow$ Initialize N-element vector of Float64

$aj\_vec \leftarrow geometry.r\_low$
$bj\_vec \leftarrow geometry.r\_high$
$cj\_vec \leftarrow geometry.z\_low$
$dj\_vec \leftarrow geometry.z\_high$
for $i = 1$ to N **do**
    $ri \leftarrow geometry.r\_middle(i)$
    $zi \leftarrow geometry.z\_middle(i)$
    $temp\_K \leftarrow$ Result of integrating $integrand\_K(\theta_{ac}, ri, zi,$ $aj\_vec, bj\_vec, cj\_vec, dj\_vec)$ with respect to θ_{ac} from 0,0 to π (using `quadgk` with `rtol=1e-6`, `atol=1e-9`)
    $K(i,:) \leftarrow temp\_K$
    $K(i,i) \leftarrow$ Result of integrating $integrand\_K\_diag(r, \theta_{ac}, ri, zi,$ $aj\_vec(i), bj\_vec(i), cj\_vec(i), dj\_vec(i))$ with respect to r from $aj\_vec(i)$ to $bj\_vec(i)$ and θ_{ac} from 0,0 to π (using `hcubature` with `rtol=1e-6`, `atol=1e-9`)
    $temp\_MBz \leftarrow$ Result of integrating $integrand\_MBz\_dist(\theta_{ac}, ri, zi,$ $aj\_vec, bj\_vec, cj\_vec, dj\_vec)$ with respect to θ_{ac} from $-\pi/2$ to $\pi/2$ (using `quadgk` with `rtol=1e-6`, `atol=1e-9`)
    $temp\_MBr \leftarrow$ Result of integrating $integrand\_MBr\_dist(\theta_{ac}, ri, zi,$ $aj\_vec, bj\_vec, cj\_vec, dj\_vec)$ with respect to θ_{ac} from $-\pi/2$ to $\pi/2$ (using `quadgk` with `rtol=1e-6`, `atol=1e-9`)
    $MBz(i,:) \leftarrow temp\_MBz$
    $MBr(i,:) \leftarrow temp\_MBr$
    $MBz(i,i) \leftarrow 0$
    $MBr(i,i) \leftarrow 0$
▷In-place unit conversions and NaN/Inf handling
$K \leftarrow$ element-wise real part of $K/10^6$
$MBz \leftarrow$ element-wise real part of $MBz/10^3$
$MBr \leftarrow$ element-wise real part of $MBr/10^3$
Replace any NaN or Inf values in K with 0,0
Replace any NaN or Inf values in MBz with 0,0
Replace any NaN or Inf values in MBr with 0,0
return K, MBr, MBz

En resumen, mientras que la implementación en `MATLAB` sigue un enfoque más rudimentario y paso a paso para el cálculo de funciones en general y de las integrales en particular, la versión en `Julia` demuestra mejoras en términos de eficiencia y claridad del código a través de la vectorización y la modularización de las funciones y cómputos. La aplicación sistemática de vectorización en la fase inicial de

migración a `Julia` constituye el primer gran paso en la mejora del rendimiento, sentando las bases para las optimizaciones posteriores de la integración de alto rendimiento en la GPU.

3.1.2. Evaluación de Resultados

Como se ha descrito en la implementación del código, el principal cambio introducido en la migración a Julia respecta al manejo de los bucles del modelo original en MATLAB. La amplia presencia de bucles explícitos y bucles anidados, así como la división del cómputo de las matrices K y M en funciones distintas introduce una gran sobrecarga que se evidencia en los tiempos de ejecución recogidos en el Cuadro 3.2. La unificación del cálculo de las matrices junto con la vectorización de las integrales permiten realizar operaciones elemento a elemento de manera concisa y altamente optimizada en único bucle.

Aunque tanto MATLAB como Julia constan de herramientas para el paralelismo, Parallel Computing Toolbox y las anotaciones @threads respectivamente, no se ha hecho un uso explícito de estos soportes. Sin embargo, Julia posee una gran ventaja, pues su compilador JIT es capaz de realizar un conjunto masivo de optimizaciones de bajo nivel como la vectorización SIMD, la eliminación de asignaciones o el inlining de funciones. Estas optimizaciones permiten que el código en Julia, incluso sin anotaciones explícitas para paralelismo de hilos y operando en un solo hilo, logre un uso altamente eficiente de la CPU y un rendimiento cercano al de lenguajes compilados.

Más allá de los paralelismos intrínsecos, Julia compila su código a código máquina nativo específico para la CPU permitiendo un acceso más directo y de bajo nivel al hardware, frente a MATLAB que interactúa a través de su entorno de ejecución. Esta sutil diferencia contribuye a la mejora de la eficiencia, principalmente en la ejecución de operaciones computacionalmente intensas como las llamadas a ‘quadgk‘ y ‘hcubature‘ en `CreationVectorPotentialMatrix12PancakeCoils` .

Otro elemento a tener en cuenta es la complejidad algorítmica del problema, que describe cómo el tiempo de ejecución de un algoritmo escala con el tamaño de su entrada. Para operaciones fundamentales como la construcción de matrices mediante iteraciones sobre sus elementos o la evaluación numérica de integrales dobles, la complejidad teórica es inherentemente cuadrática, denotándose como $O(N^2)$ [21]. Este hecho implica que, en un escenario ideal y asintótico, si el tamaño de la entrada N se duplica, el tiempo de ejecución se multiplicaría por $2^2 = 4$. La función `CreationVectorPotentialMatrix12PancakeCoils`, por ejemplo, implementa un bucle anidado, iterando sobre N elementos, para los cuales realiza operaciones que dependen de N elementos adicionales. Sin embargo, aunque en el código se manifieste la complejidad $O(N^2)$, al comparar los tiempos de computación observados en MATLAB para la operación `K & M Matrix` con diferentes

geometrías, se aprecian matices en este comportamiento.

Al incrementar el tamaño de la entrada N alterando la geometría de un doble pancake con 20 espiras (20,2) a dos dobles pancakes con 40 vueltas (40,2), se observa un aumento de N en un factor de 4. Teóricamente, esto implicaría que el tiempo de ejecución debería multiplicarse por $4^2 = 16$. Sin embargo, el aumento real del tiempo de ejecución en MATLAB para estas operación es de aproximadamente 7.56 veces (pasando de 926.34 s a 7007.5 s) para la geometría (40,2) y de 23.96x para el teórico incremento de ≈ 51.84 que debería darse para la geometría (144,2). Esta discrepancia entre el factor de crecimiento observado en los tiempos de ejecución ($\approx$ 7.56x y $\approx$ 23.96x) y el teóricamente esperado para un comportamiento $O(N^2)$ (16x y 52x) sugiere que, aunque la naturaleza fundamental del algoritmo es cuadrática, la implementación en MATLAB exhibe un rendimiento más favorable en estos rangos de entrada.

Contemplando los tiempos de ejecución de Julia se observa que en este caso los factores de crecimiento sí se corresponden con la complejidad algorítmica $O(N^2)$. Realizando la comparativa con las distintas geometrías se establece que la transición de (20,2) a (40,4) se traduce en un incremento del tiempo de computación de las matrices K y M en un factor de 17,3x, y para la geometría (144,2) de 58.9x. Esto confirma que, si bien el tiempo de computación global se reduce, la naturaleza cuadrática del algoritmo en Julia sí se mantiene, pues los valores teóricos para los factores de crecimiento de las geometrías (40,4) y (144,2) respecto de la geometría base (20,2) deberían ser 16x y 51,84x respectivamente.

Cuadro 3.1: Comparación de Características de Rendimiento por Lenguaje

Característica	**MATLAB**	**Julia**
Manejo de bucles	Bucle anidado	Vectorización
Paralelismo	Nulo	SIMD e inlining
Tiempo de ejecución	Alto	Medio
Acceso a hardware	CPU	CPU (compilación código nativo)
Complejidad algorítmica	$O(N^2)$ (mejorada)	$O(N^2)$

Un análisis más detallado de los tiempos de ejecución revela que, si bien todas las funciones experimentan una mejora en la eficiencia de cómputo al pasar de MATLAB a Julia, excepto quizá en la función de simetría que a veces se ralentiza ligeramente, el impacto de esta optimización no es uniforme en todas las operaciones. Las funciones de menor calibre, como $SetMeshSuperconductor2PancakeCoils$ o $BuildGeometry12PancakeCoils$, aunque muestran mejoras relativas significativas (por ejemplo, 'BuildGeometry12PancakeCoils' pasa de 0.0020 s en MATLAB a 0.000005 s en Julia para (20,2), una mejora del 97.28 %), sus tiempos absolutos

son tan reducidos que su contribución al tiempo de ejecución global es despreciable. El punto diferencial en la eficiencia se observa de manera contundente en el cómputo de las matrices `K & M`, que representa la carga computacional dominante. Con 7007.5 s en MATLAB para la geometría (40,4), experimenta una reducción sustancial a 2091.81 s en Julia para la misma geometría, lo que constituye una mejora del 70.15 %. Para la geometría (20,2) esta reducción del tiempo es aún superior alcanzando un 87.20 %, comportamiento que se traslada a la geometría (144,2) con una mejora del 68.11 %. Es por tanto en esta función concreta donde el traslado a Julia se traduce en un impacto más significativo y determinante en el rendimiento general del sistema.

Otro de los puntos diferenciales, que abarca junto con las matrices K y M el 100 % del tiempo de cómputo es la resolución de ecuaciones diferenciales. El solver de ODE de Julia $ROS34PW2$ resulta ser extremadamente eficiente en comparación con la implementación en MATLAB dando lugar a unas ganancias en tiempo superiores al 60 % para todas las geometrías. El gran requerimiento temporal de los solvers convertían a este elemento en la segunda parte más crítica para lograr una incremento significativo en la eficiencia del modelo. Además, cabe resaltar que la energía total disipada en el sistema Q es prácticamente equivalente entre los modelos, lo cual indica que este primer modelo logra replicar el comportamiento de MATLAB sin detrimento de la precisión para cualquiera de las geometrías. En términos generales, el Modelo I alcanza una mejora total para cualquier geometría en la construcción de matrices en torno al 70 %, y un incremento de la eficiencia en la convergencia de los ODE solvers del 60 %. No obstante, para poder obtener resultados en intervalos cortos de tiempo para casos de uso reales es necesario seguir reduciendo el tiempo de cómputo de las matrices K y M.

Analizando las gráficas de Potencia y de J/J_c en las Figuras 3.2 a 3.4 se aprecia que la distinción visual entre las distribuciones de potencia y corriente obtenidas con Julia, en comparación con las de MATLAB, es mínimamente perceptible. Se produce una divergencia ligeramente más pronunciada en las gráfica de distribución de calor, donde la tonalidad es más ligera dificultando su visualización en contraste con la obtenida mediante MATLAB. La equivalencia entre las gráficas extraídas en MATLAB y en Julia viene avalada por los valores de potencia disipada Q, que como muestra la Tabla 3.2 son consistentes y análogos para todas las geometrías.

Cuadro 3.2: Tiempos de Ejecución para Diferentes Geometrías en MATLAB y Julia.

(a) Geometría (20,2)

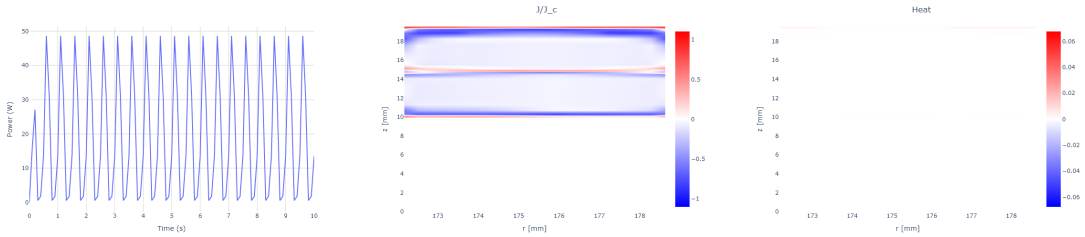
Característica	**MATLAB**	**Julia CPU**	**Δ CPU [%]**
Mesh [s]	0.0184	0.0005	97.28
Geometry [s]	0.0020	0.000005	99.75
K Matrix [s]	258.51	–	–
M Matrix [s]	667.83	–	–
K & M Matrix [s]	926.34	120.18	87.02
ϕ Matrix [s]	0.0775	0.0035	95.48
Symmetry [s]	0.0353	0.0168	52.40
Solver [s]	400.92	60.60	84.88
Q [W]	186.5815	186.64	–

(b) Geometría (40,4)

Característica	**MATLAB**	**Julia CPU**	**Δ CPU [%]**
Mesh [s]	0.0882	0.0117	86.73
Geometry [s]	0.0024	0.00004	98.33
K Matrix [s]	2193.1	–	–
M Matrix [s]	4814.4	–	–
K & M Matrix [s]	7007.5	2091.81	70.15
ϕ Matrix [s]	6.5275	0.0109	99.88
Symmetry [s]	0.2025	0.4497	-122.07
Solver [s]	4923.20	1922.40	60.95
Q [W]	2171.00	2171.11	–

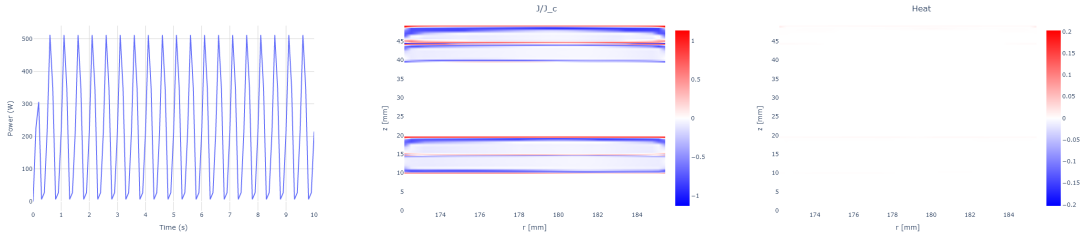
(c) Geometría (144,2)

Característica	**MATLAB**	**Julia CPU**	**Δ CPU [%]**
Mesh [s]	0.1418	0.1028	27.50
Geometry [s]	0.0030	0.000001	99.97
K Matrix [s]	6863.9	–	–
M Matrix [s]	15334	–	–
K & M Matrix [s]	22197.9	7079.44	68.11
ϕ Matrix [s]	48.9914	0.0359	99.97
Symmetry [s]	0.9235	2.7097	-193.41
Solver [s]	27425	10801.74	60.61
Q [W]	12289	12293.35	–



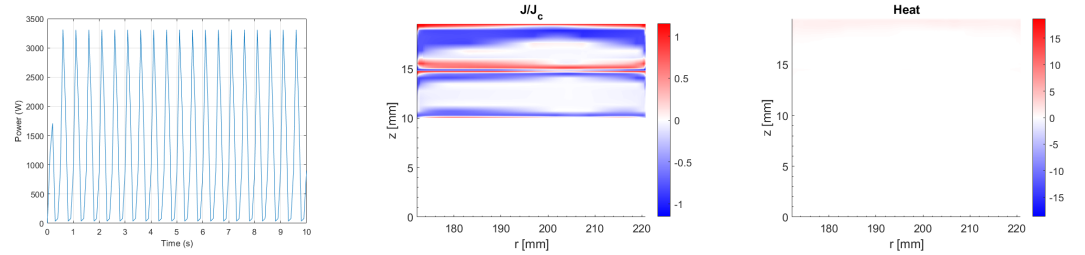
(a) Potencia [W] (b) J/J_c (c) Calor

Figura 3.2: Distribuciones en Julia sobre CPU para la Geometría (20,2).



(a) Potencia [W] (b) J/J_c (c) Calor

Figura 3.3: Distribuciones en Julia sobre CPU para la Geometría (40,4).



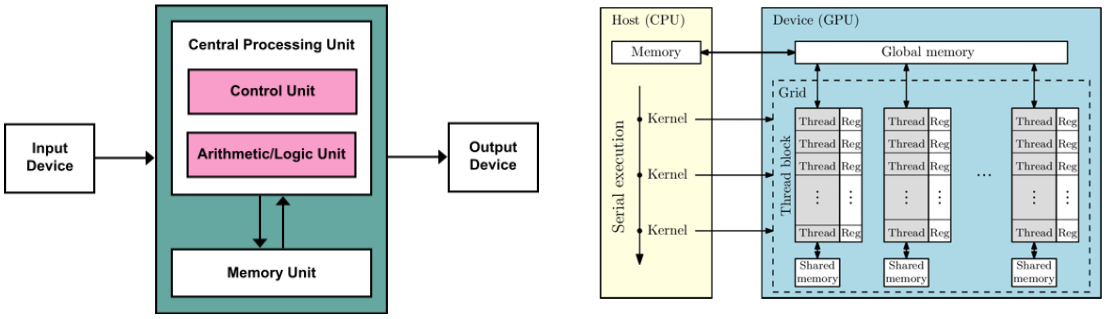
(a) Potencia [W] (b) J/J_c (c) Calor

Figura 3.4: Distribuciones en Julia sobre CPU para la Geometría (144,2).

3.2. Modelo II: Traslado de computación K Y M a GPU

El traslado de la ejecución de un modelo desde la **CPU** a la **GPU** es una estrategia común en la computación moderna para aplicaciones que requieren un alto rendimiento y procesamiento de grandes volúmenes de datos. Las CPUs han sido el cerebro fundamental de los sistemas informáticos desde los inicios de la computación, responsabilizándose de la ejecución del sistema operativo, la gestión de la memoria principal y la manipulación del espacio en disco [22]. La arquitectura interna de una CPU incluye una **Unidad de Control (Control Unit)**, que coordina las instrucciones y el flujo de datos entre la CPU y otros componentes del sistema, como la memoria principal (*Main Memory*). Asimismo, la **Unidad Lógico Aritmética (ALU - Arithmetic Logic Unit)** es el circuito interno encargado de realizar todas las operaciones aritméticas y lógicas y recibe tres tipos de entradas: señales de control de la Unidad de Control, datos de la memoria para las operaciones, y la información de estado de operaciones previas. La Figura 3.5a ilustra a alto nivel la arquitectura descrita. A pesar de su diseño que convierte a las CPUs en excepcional unidades para el procesamiento secuencial eficiente, su arquitectura las convierte en menos adecuada para la ejecución masiva de un gran número de operaciones paralelas e idénticas.

Para este tipo de aplicaciones computacionalmente intensas, la GPU, originialmente concebida para renderizar gráficos, suele ser la herramienta por defecto al lograr eficiencias muy superiores a la CPU por la configuración de su estructura [22], mostrada en la Figura 3.5b. Estos procesadores de **múltiples núcleos** son gestionados desde la CPU, el sistema anfitrión, mediante el envío de "kernels", es decir, de funciones a ejecutar en la GPU. Esta está diseñada con una arquitectura masivamente paralela que incluye múltiples **Streaming Multiprocessors (SMs)**, una especie de "supervisores"de grupos de trabajadores que son los núcleos CUDA o *threads*. Cada hilo tiene acceso a su propia memoria local (*Register*) para almacenar variables temporales. Además, los SMs tienen acceso a la *Shared memory*, que actúa como una caché y es accesible por cualquier núcleo CUDA dentro de ese SM, siendo significativamente más rápida que la *Global memory*. La capacidad de ejecutar simultáneamente miles de tareas simples, comúnmente conocido como **computación paralela**, convierte a las GPUs en una opción ideal para modelos que requieren un alto rendimiento y el procesamiento de grandes volúmenes de datos mediante operaciones simples y repetitivas, superando las limitaciones de la CPU en estos escenarios.



(a) CPU [23] (b) GPU [24]

Figura 3.5: Funcionamiento Básico de las principales Unidades de Procesamiento.

A continuación, se detallan las principales razones por las que justificaríamos el traslado del cálculo de las matrices K y M del Modelo I de la CPU a la GPU:

- **Aceleración Significativa del Rendimiento:** La principal razón para migrar a la GPU es la **drástica mejora en el rendimiento** para modelos basados en extensas operaciones matriciales, transformaciones de datos, simulaciones numéricas o entrenamiento de algoritmos de Deep Learning [25]. Mientras que una CPU puede manejar un número limitado de hilos, una GPU puede ejecutar miles de hilos simultáneamente, lo cual resulta en una aceleración de varios órdenes de magnitud en el tiempo de ejecución para tareas computacionalmente intensivas.

- **Optimización para Cargas de Trabajo Paralelas:** Operaciones como la multiplicación de matrices, o las integrales dobles y triples, que son comunes en muchos modelos científicos y en este modelo SMES en particular, pueden descomponerse en miles de pequeñas operaciones independientes que se ejecutan en paralelo. La arquitectura de la GPU, que permite manejar múltiples flujos de datos concurrentemente, es perfecta para procesar este tipo de cargas de trabajo, al poder realizar de manera simultáne miles de estas operaciones repetitivas en hilos paralelos. Al mover el modelo a la GPU se optimiza la utilización de recursos y se logra reducir el tiempo total de computación.

- **Escalabilidad para Modelos de Gran Escala:** A medida que los modelos se vuelven más grandes y complejos (simulaciones u operaciones báscias sobre miles de puntos de datos), la capacidad de una CPU para manejarlos se ve rápidamente superada. Las GPUs ofrecen una **escalabilidad superior** para estos escenarios, permitiendo procesar volúmenes de datos que a nivel temporal se vuelve inviable en una CPU. Este hecho se evidencia en los

tiempos de ejecución del Modelo I 3.2, donde los ligeros aumentos en la geometría del problema, sin incurrir en grandes complejidades, requieren ejecuciones prolongadas a lo largo de varias horas.

3.2.1. Traslado de Integrales K y M a GPU

Tras confirmar la operatividad y la correcta funcionalidad del Modelo I implementado en Julia, se procedió a realizar un análisis de rendimiento para identificar los cuellos de botella computacionales. Para ello se emplearon herramientas de *profiling* específicas de cada entorno de desarrollo: la macro `@time` en Julia y las funciones `tic/toc` en MATLAB. Estas herramientas de medición permiten generar perfiles de rendimiento detallados cuantificando en segundos el tiempo de ejecución requerido por cada una de las funciones que componen el modelo.

Como se evidencia claramente en el Cuadro 3.2, el principal cuello de botella del modelo reside en el cálculo de las matrices K y M, que consumen prácticamente la totalidad del tiempo de ejecución global del modelo. Puesto que el cálculo de las integrales supone una elevada reiteración de los mismos cálculos la capacidad de cómputo en paralelo de las GPU supone una herramienta ideal para acelerar su ejecución.

Para alcanzar tiempos de ejecución más razonables independientemente del tamaño y complejidad del modelo, la función `CreationPotentialMagneticMatrix12PancakeCoils_CUDA` representa una evolución significativa respecto a su predecesora, implementando una paralelización masiva a través de la arquitectura CUDA de NVIDIA. El objetivo principal de esta función es la construcción eficiente de las matrices de potencial vectorial K y de campo magnético MBr y MBz para un sistema de bobinas tipo *pancake* transicionando de un paradigma de cómputo secuencial a un modelo de cómputo paralelo. Mientras que las implementaciones previas realizaban las integrales numéricas mediante bucles iterativos o llamadas a funciones de integración de alto nivel sobre la CPU, esta nueva función se centra en la aceleración por hardware. Para poder trabajar en este entorno es necesario transferir los datos geométricos de las bobinas de la memoria principal a la memoria de la GPU, lo cual minimiza la costosa transferencia de datos entre CPU y GPU durante los cálculos. Las matrices de resultado (K, MBr, MBz) también se inicializan directamente en la memoria de la GPU para evitar dichas transferencias a posterioro.

El elemento central de la optimización está compuesto por la función `cuda_kernel`, un programa que se ejecuta de forma concurrente por miles de hilos en la GPU, donde cada hilo es responsable de calcular un elemento específico de las matrices de salida. A diferencia de las versiones anteriores en la CPU, el kernel CUDA implementa una discretización y suma explícita de Riemann para las integrales angulares. Para ello, se definen rangos de ángulos y sus correspondientes pasos

de discretización. La elección del número de puntos de integración y la asignación dinámica de estos basados en regiones en la matriz se fundamentan en una estrategia para balancear la precisión con el rendimiento, ajustándose a la complejidad de la función en ciertas zonas. Durante la transición a la GPU se identificó que con pocos puntos de integración existen secciones dentro de las matrices que divergen notablemente de sus valores objetivo. Dichas secciones se muestran de manera gráfica en la Figura 3.6. No se incluye la matriz M debido a la similitud en apariencia, con la única distinción de que carece del error en la diagonal principal y de que el error general es superior en magnitud.

Dentro del kernel, las operaciones se formulan para ser altamente paralelizadas, y aunque Julia por sí misma favorece la vectorización, aquí la paralelización es a un nivel aún más granular, siendo orquestada directamente por CUDA. Cada hilo calcula un par (i, j) específico, permitiendo la evaluación simultánea de miles de elementos de la matriz. La inclusión de una ϵ y el uso de funciones `max()` y `min()` en los denominadores y argumentos de `log` son salvaguardas que previenen divisiones por cero, logaritmos de cero o números negativos, y desbordamientos, comunes en cálculos físicos con posibles singularidades o valores extremos.

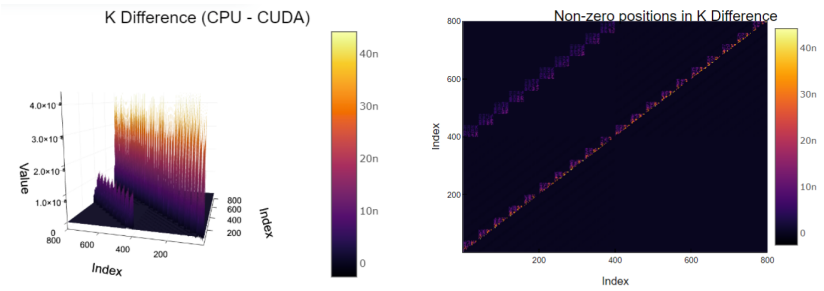


Figura 3.6: Diferencia en K entre modelo de Julia en CPU y en CUDA con pocos puntos de integración.

Un elemento a destacar de esta implementación es la excepción para los elementos diagonales de la matriz K, donde la función recurre a la operación `hcubature` en la CPU. La integral para los elementos diagonales (cuando $i = j$) es numéricamente más desafiante para la aproximación de Riemann con discretización fija en el kernel CUDA, resultando en unos valores notablemente desviados de los valores reales que deberían obteners. Debido a que `hcubature` únicamente se calcula sobre los elementos de la diagonal principal, aplicar el cálculo sobre la CPU ofrece una solución más eficiente y robusta para este caso específico, que incrementar en varios órdenes de magnitud los puntos de integración necesarios para alcanzar un error aceptable de estos valores. Finalmente, todos los resultados se transfieren de nuevo a la CPU y se escalan, y se realiza una limpieza de valores `NaN` e `Inf` para

asegurar la estabilidad numérica del resultado.

Por último, mencioar que en Julia la convención de añadir un '¡ al final del nombre de una función, como en `cuda_kernel!`, significa que la función modifica destructivamente uno o más de sus argumentos de entrada. Es decir, esta convención indica que la función opera ïn-place"sobre las matrices `K`, `MBr`, y `MBz` que se le pasan, en lugar de devolver nuevas matrices, siendo esta práctica fundamental en programación de alto rendimiento y en el desarrollo para GPU, para evitar la asignación de memoria adicional y la copia de datos innecesaria.

Algorithm 6 K and M Matrix Calculations with Julia using CUDA

Function: CreationVectorPotentialMatrix12PancakeCoils_CUDA(*geometry*)
Input:
  geometry: A structure containing geometric parameters, including:
    $geometry.r\_middle$: Radial midpoints.
    $geometry.z\_middle$: Axial midpoints.
    $geometry.r\_low$: Lower radial bounds for integration (a_j).
    $geometry.r\_high$: Upper radial bounds for integration (b_j).
    $geometry.z\_low$: Lower axial bounds for integration (c_j).
    $geometry.z\_high$: Upper axial bounds for integration (d_j).
Output:
  K: Vector potential matrix.
  MBr: Radial component of the magnetic field matrix.
  MBz: Axial component of the magnetic field matrix.
▷ Transfer geometry data to GPU
$ri \leftarrow$ CuArray(reshape($geometry.r\_middle, :, 1$))
$zi \leftarrow$ CuArray(reshape($geometry.z\_middle, :, 1$))
$aj \leftarrow$ CuArray(reshape($geometry.r\_low, 1, :$))
$bj \leftarrow$ CuArray(reshape($geometry.r\_high, 1, :$))

19: $cj \leftarrow$ CuArray(reshape($geometry.z\_low, 1, :$))
20: $dj \leftarrow$ CuArray(reshape($geometry.z\_high, 1, :$))
21: ▷ Initialize output matrices on GPU
22: $K_{gpu} \leftarrow$ CUDA.zeros(Float64, N, N)
23: $MBr_{gpu} \leftarrow$ CUDA.zeros(Float64, N, N)
24: $MBz_{gpu} \leftarrow$ CUDA.zeros(Float64, N, N)
25: ▷ Define integration points and transfer to GPU
26: $n\_points\_low \leftarrow 1000$
27: $n\_points\_high\_K \leftarrow 2000$
28: $n\_points\_high\_M \leftarrow 5000$
29: $\theta\_range\_K\_low \leftarrow$ CuArray(collect(range($0, \pi, length = n\_points\_low$)))
30: $\theta\_range\_M\_low \leftarrow$ CuArray(collect(range($-\pi/2, \pi/2, length = n\_points\_low$)))
31: $\theta\_range\_K\_high \leftarrow$ CuArray(collect(range($0, \pi, length = n\_points\_high\_K$)))
32: $\theta\_range\_M\_high \leftarrow$ CuArray(collect(range($-\pi/2, \pi/2, length = n\_points\_high\_M$)))
33: $d\theta\_K\_low \leftarrow \pi / n\_points\_low$
34: $d\theta\_M\_low \leftarrow \pi / n\_points\_low$
35: $d\theta\_K\_high \leftarrow \pi / n\_points\_high\_K$
36: $d\theta\_M\_high \leftarrow \pi / n\_points\_high\_M$
37: $n\_points\_r \leftarrow 1000$
38: $r\_ranges\_gpu \leftarrow$ create_r_ranges_gpu(aj, bj, n_points_r)
39: $dr \leftarrow$ CuArray($(bj. - aj)./n\_points\_r$)
40: ▷ Define regions for high number of integration points
41: $region1\_start \leftarrow 1$, $region1\_end \leftarrow Ne * Nz * Nr * Nc/2$
42: $region2\_start \leftarrow Ne * Nz * Nr * Nc/2 + 1$, $region2\_end \leftarrow Ne * Nz * Nr * Nc$
43: ▷ Define and launch the CUDA kernel
44: **procedure** CUDA_KERNEL!($K_{gpu}, MBr_{gpu}, MBz_{gpu}, \dots$)
45: ▷ Input/output parameters as in source code
46: $(idx, idy) \leftarrow$ thread and block indices
47: **if** $idx \leq N$ and $idy \leq N$ **then**
48: $sum\_K \leftarrow 0{,}0$, $sum\_MBz \leftarrow 0{,}0$, $sum\_MBr \leftarrow 0{,}0$
49: $epsilon \leftarrow 1{,}0e-10$
50: ▷ Select number of integration points for K
51: **if** $((region1\_start \leq idx \leq region1\_end \wedge region1\_start \leq idy \leq region1\_end) \vee (region2\_start \leq idx \leq region2\_end \wedge region2\_start \leq idy \leq region2\_end))$ **then**
52: $\theta\_range\_K \leftarrow \theta\_range\_K\_high$, $d\theta\_K \leftarrow d\theta\_K\_high$
53: **else**
54: $\theta\_range\_K \leftarrow \theta\_range\_K\_low$, $d\theta\_K \leftarrow d\theta\_K\_low$

for $k = 1$ to length(θ_range_K) **do**
  $\theta \leftarrow \theta\_range\_K[k]$
  $ej \leftarrow (aj[idy] + bj[idy]) * 0{,}5$
  ▷ Calculate *term1*, *term2*, *term3* for K with 'safe_log' logic
  $sum\_K \leftarrow sum\_K + (bj[idy] - aj[idy])/6{,}0 \cdot (\text{term1} + \text{term2} + \text{term3}) \cdot d\theta\_K$
$K_{gpu}[idx, idy] \leftarrow sum\_K$
▷ Select number of integration points for M
if $((region1\_start \leq idx \leq region1\_end \wedge region1\_start \leq idy \leq region1\_end) \vee (region2\_start \leq idx \leq region2\_end \wedge region2\_start \leq idy \leq region2\_end))$ **then**
  $\theta\_range\_M \leftarrow \theta\_range\_M\_high$, $d\theta\_M \leftarrow d\theta\_M\_high$
else
  $\theta\_range\_M \leftarrow \theta\_range\_M\_low$, $d\theta\_M \leftarrow d\theta\_M\_low$
for $k = 1$ to length(θ_range_M) **do**
  $\theta \leftarrow \theta\_range\_M[k]$
  $ej \leftarrow (aj[idy] + bj[idy]) * 0{,}5$
  ▷ Calculate $term\_MBz$ and $term\_MBr$ with safeguard logic
  $sum\_MBz \leftarrow sum\_MBz + \text{term\_MBz} \cdot d\theta\_M$
  $sum\_MBr \leftarrow sum\_MBr + \text{term\_MBr} \cdot d\theta\_M$
$MBz_{gpu}[idx, idy] \leftarrow sum\_MBz$
$MBr_{gpu}[idx, idy] \leftarrow sum\_MBr$
▷ Configure and launch the CUDA kernel
$max\_threads \leftarrow 256$
$threads\_dim \leftarrow \text{mín}(32, \lceil \text{sqrt}(max\_threads) \rceil)$
$threads\_per\_block \leftarrow (threads\_dim, threads\_dim)$
$blocks\_per\_grid \leftarrow (\lceil N/threads\_dim \rceil, \lceil N/threads\_dim \rceil)$
CUDA_KERNEL!(args) ▷ Launch kernel on GPU with parameters
Synchronize CUDA
▷ Transfer results from GPU to CPU and scale
$K_{cpu} \leftarrow \text{Array}(K_{gpu})$
$MBr_{cpu} \leftarrow \text{Array}(MBr_{gpu})/10^3$
$MBz_{cpu} \leftarrow \text{Array}(MBz_{gpu})/10^3$
▷ Perform `hcubature` operation on diagonal elements of K (on CPU)
for $i = 1$ to N **do**
  $K_{cpu}[i, i] \leftarrow \text{hcubature}(\text{integrand\_K\_diag}, \dots)[1]$
  $MBr_{cpu}[i, i] \leftarrow 0$
  $MBz_{cpu}[i, i] \leftarrow 0$
$K_{cpu} \leftarrow K_{cpu}/10^6$
Replace any NaN or Inf values in K_{cpu}, MBr_{cpu}, MBz_{cpu} with 0,0
return $K_{cpu}, MBr_{cpu}, MBz_{cpu}$

3.2.2. Evaluación de Resultados

El objetivo principal de migrar el cómputo de las matrices K y M a la GPU es lograr una aceleración del proceso sin comprometer la precisión de los resultados. Mientras que para el modelo en CPU se emplea un único hilo de procesamiento secuencial, los modelos de GPU despliegan 256 hilos de ejecución concurrentes por bloques (16x16 hilos), permitiendo paralelizar masivamente los cálculos e integrales de K y de M. La Tabla 3.3 presenta una comparación de características de rendimiento entre la implementación de Julia en CPU (Modelo I) y la versión acelerada por GPU (Modelo II) para el cálculo de estas matrices.

Una distinción fundamental entre ambos modelos radica en el **manejo de los bucles**. Mientras que la implementación en CPU de Julia se beneficia de la vectorización de sus cálculos, que permite suprimir los bucles anidados de MATLAB e implementar operaciones eficientes elemento a elemento, la versión en GPU se centra en la **paralelización masiva**. En el contexto de la GPU, el uso de bucles explícitos, como el que se observa en 6 para la integración de la matriz K (`for k in 1:length(`θ`_range_K)`), no implica una ejecución secuencial ineficiente. Estos bucles son internamente paralelizados a través de miles de hilos de procesamiento, donde cada hilo se encarga de una parte de la integración o de un elemento de la matriz en paralelo. Debido a que las operaciones involucradas en el cálculo de estas matrices (integrales, raíces cuadradas y logaritmos) deben realizarse sobre una multitud de puntos, la arquitectura de la GPU, diseñada para el procesamiento simultáneo de un gran número de tareas idénticas, ofrece una ventaja sustancial para acelerar los tiempos de computación.

Aunque Julia en CPU ya posee capacidades avanzadas, como la compilación a código nativo y la capacidad de optimizar masivamente el código a través de optimizaciones SIMD incluso sin anotaciones explícitas, la complejidad y escala del sistema actual superan las capacidades de paralelismo en CPU para lograr una reducción temporal aún mayor a la ya conseguida. Puesto que el problema requiere miles de computaciones de puntos independientes, la GPU ejecuta estas operaciones simultáneamente, trabajando cada hilo una porción del bucle de integración de manera concurrente. En el bucle de integración para K, por ejemplo, cada hilo de la GPU puede calcular los términos `term1_num_sq`, `term1_num`, `term1_den_sq`, etc., para un par `(idx, idy)` específico de la matriz de salida, aprovechando al máximo el procesamiento paralelo.

Un elemento a destacar es que la precisión de la integración numérica es un factor crítico para la convergencia de los solvers. En este estudio, se observó que la implementación de un elevado número de puntos de integración es indispensable para mitigar las pequeñas imprecisiones de la computación en la GPU que, de otro modo, impiden la obtención de resultados concluyentes. Para abordar esto, se llevaron a cabo dos simulaciones diferenciadas. En la primera, referida como GPU

I, se especificaron 2000 puntos de integración para las regiones con pocos puntos de las matrices K y M, y para la región densa de K, mientras que para la región densa de M se emplearon 4000 puntos, dada la mayor sensibilidad al error de la matriz M. Adicionalmente, para evaluar el equilibrio entre la precisión numérica y la demanda computacional, una segunda simulación aplica uniformemente 4000 puntos a todas las regiones de integración.

Para la computación en la CPU con Julia se corroboró que el factor de crecimiento del modelo se comportaba de acuerdo con la complejidad algorítmica $O(N^2)$. Al sustituir la geometría (20,2) por (40,4) y por (144,2) el tiempo de computación de las matrices K y M se incrementaba en un factor de 17,3x y 58.9x respectivamente, muy cercanos a los teóricos 16x y 52x que deberían presentar. En el caso de la GPU esta comprobación se va a realizar para cada uno de los casos individualmente, donde el modelo con un número de puntos de integración variable GPU I experimenta un incremento del tiempo por un factor de $\approx$ 15.85 para (40,4) y de $\approx$ 50.33x para (144,2), y el modelo GPU II sufre una subida de 16.22x y de 51.70x para cada una de las geometrías. Como evidencian los valores, a pesar de las mejoras temporales la complejidad algorítmica se traslada a los tiempos de computación que evolucionan paralelamente con la complejidad de la geometría.

Cuadro 3.3: Comparación de Características de Rendimiento (Modelo I (Julia CPU) vs. Modelo II (Julia con CUDA)

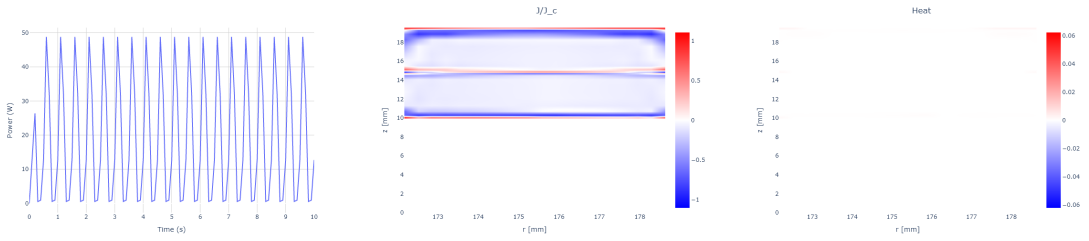
Característica	**Julia CPU**	**Julia GPU**
Manejo de bucles	Vectorización	Paralelización
Paralelismo	SIMD e inlining	CUDA
Tiempo de ejecución	Medio	Bajo
Acceso a hardware	CPU	GPU
Complejidad algorítmica	$O(N^2)$	$O(N^2)$

Un análisis más en detalle de los tiempos de ejecución, en concreto de las matrices K y M revela que, la migración a la GPU implica una mejora en la eficiencia de cómputo, si bien es necesario tomar una decisión sobre el balance que se desea entre precisión y eficiencia temporal. Un mayor número de puntos de integración alcanza valores de potencia más cercanos al valor real, 2117.89 W frente a los 2171.50 de la CPU, mientras que una disminución en los puntos de integración implica una reducción de la precisión, alcanzando el modelo GPU I solo 2069.63 W para la geometría (40,4). Sin embargo, con esa geometría el modelo GPU I alcanza una mejora de aproximadamente el 60 % en tiempo de computación frente al 23 % alcanzado por el modelo GPU II. Se produce una situación equivalente para la geometría (144,2), mientras que para el caso de un doble pancake con 20

espiras el cambio en la precisión es imperceptible, aunque el tiempo de cómputo para las matrices K y M supone casi la mitad para el caso GPU I frente al caso GPU II. Se puede afirmar, por lo tanto, que la migración a la GPU, excepto que se opte por un número desorbitado de puntos de integración sí supone una mejora en la eficiencia de cómputo de las matrices, si bien está supeditada al nivel de precisión requerido. La reducción temporal es lógicamente menos pronunciada que en el Modelo I, pero supone un paso adicional en la obtención de un modelo capaz de simular un sistema SMES con tiempos de respuesta reducidos.

Otro elemento a tener en cuenta es el tiempo de cómputo de los solvers, que se resiente conforme menor es el número de puntos de integración, pues por norma general se trata de números ligeramente distintos y más imprecisos, lo cual ralentiza la convergencia. Este hecho viene avalado por la pérdida en tiempo de convergencia de los solvers respecto al modelo implementado sobre la CPU de Julia. No obstante, se sigue tratando de un rendimiento superior al del solver de MATLAB, alcanzando reiteradamente una mejor precisión en los dos elementos críticos del modelo. Las divergencias que se observan en el resto de funciones simplemente se deben a que los tiempos de ejecución de un mismo modelo no son constantes y las pequeñas divergencias se pueden deber a otros procesos corriendo simultáneamente en el ordenador o la variabilidad de los entornos de ejecución.

Las gráficas de Potencia y de J/J_c en las Figuras 3.7 a 3.12 muestran las simulaciones para los modelos GPU I y GPU II con geometrías (20,2), (40,4) y (144,2). Analizando las representaciones a penas se identifican diferencias, indicando que independientemente de los puntos de integración y de su distribución, con niveles de precisión razonables, como los tratados en este caso, visualmente no se evidencian las leves imprecisiones del modelo. Quizá en las gráfica de distribución de calor es la divergencia entre la precisión de los modelos ligeramente más evidente a través de una concentración sutilmente más marcada en el modelo GPU II, que presenta una mayor precisión. A todos los efectos no se puede concluir que exista una diferencia significativa entre los modelos GPU I y GPU II.



(a) Potencia [W] (b) J/J_c (c) Calor

Figura 3.7: Distribuciones en Julia sobre GPU para la Geometría (20,2) con Puntos Variables de Integración según región GPU I.

Cuadro 3.4: Tiempos de Ejecución para Diferentes Geometrías en Julia (CPU y GPU).

(a) Geometría (20,2)

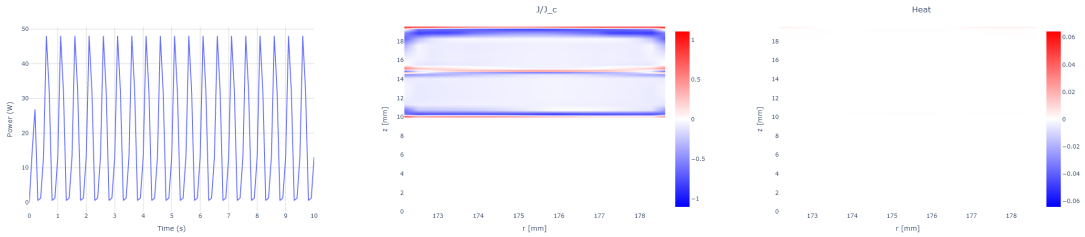
Característica	**CPU**	**GPU I**	**Δ GPU I [%]**	**GPU II**	**Δ GPU II [%]**
Mesh [s]	0.0005	0.00158	-216.00	0.000393	21.40
Geometry [s]	0.000005	0.0000013	74.00	0.000001	80.00
K & M Matrix [s]	120.18	55.23	54.04	99.42	17.30
ϕ Matrix [s]	0.0035	0.00205	41.43	0.00558	-59.43
Symmetry [s]	0.0168	0.0034	79.76	0.0044	73.81
Solver [s]	60.60	106.22	-75.28	73.70	-21.62
Q [W]	186.64	185.50	–	185.45	–

(b) Geometría (40,4)

Característica	**CPU**	**GPU I**	**Δ GPU I [%]**	**GPU II**	**Δ GPU II [%]**
Mesh [s]	0.0117	0.0071	39.32	0.0098	16.24
Geometry [s]	0.00004	0.000001	97.50	0.000001	97.50
K & M Matrix [s]	2091.81	875.34	58.15	1613.09	22.89
ϕ Matrix [s]	0.0109	0.0081	25.70	0.0717	-556.00
Symmetry [s]	0.4497	0.8084	-79.76	0.8395	-86.68
Solver [s]	1922.4	3388.68	-76.27	2459.63	-27.94
Q [W]	2171.11	2069.52	–	2117.89	–

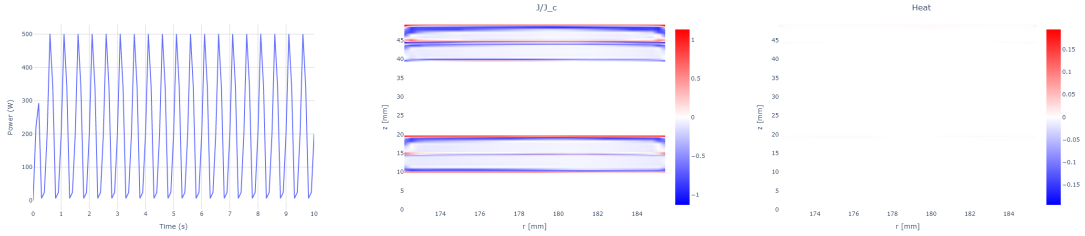
(c) Geometría (144,2)

Característica	**CPU**	**GPU I**	**Δ GPU I [%]**	**GPU II**	**Δ GPU II [%]**
Mesh [s]	0.1028	0.0156	84.82	0.0173	83.17
Geometry [s]	0.000001	0.000001	0.0	0.000001	0.0
K & M Matrix [s]	7079.44	2779.77	60.73	5140.93	27.38
ϕ Matrix [s]	0.0359	0.0157	56.26	0.0151	57.93
Symmetry [s]	2.7097	2.5917	4.35	25.6117	-89.42
Solver [s]	10801.74	16624.24	-53.90	11895.14	-10.12
Q [W]	12293.35	12032.49	–	12135.48	–



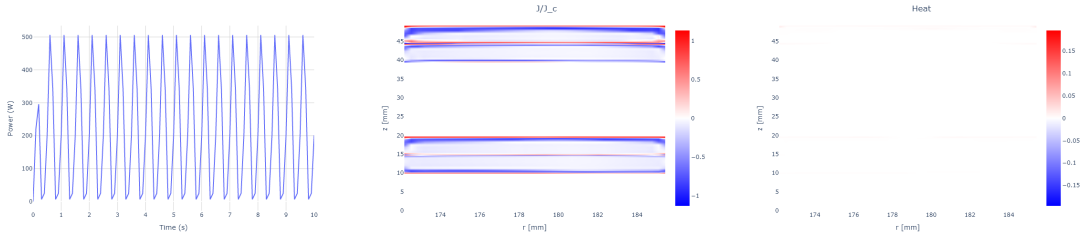
(a) Potencia [W] (b) J/J_c (c) Calor

Figura 3.8: Distribuciones en Julia sobre GPU para la Geometría (20,2) con 4000 Puntos Fijos de Integración para todas las regiones GPU II.



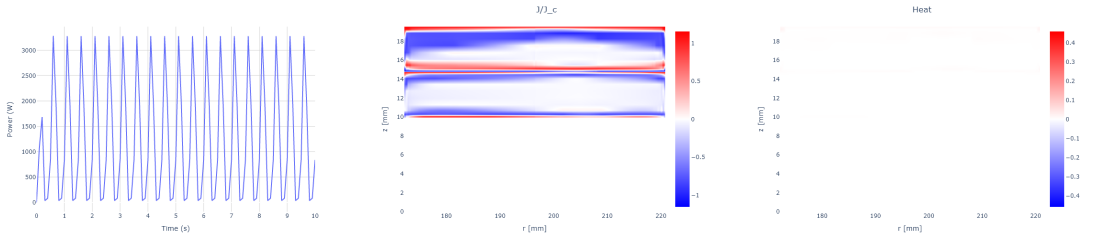
(a) Potencia [W] (b) J/J_c (c) Calor

Figura 3.9: Distribuciones en Julia sobre GPU para la Geometría (40,4) con Puntos Variables de Integración según región GPU I.



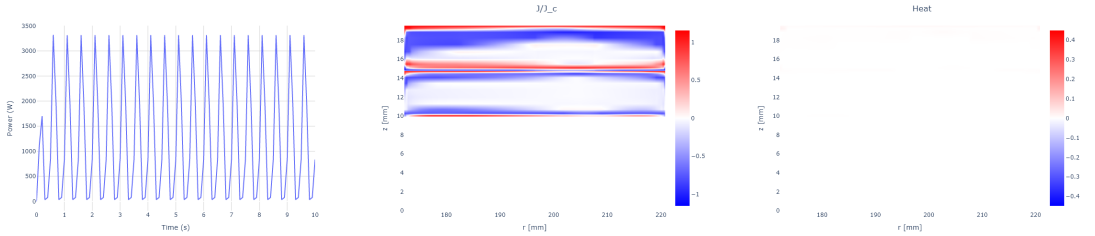
(a) Potencia [W] (b) J/J_c (c) Calor

Figura 3.10: Distribuciones en Julia sobre GPU para la Geometría (40,4) con 4000 Puntos Fijos de Integración para todas las regiones GPU II.



(a) Potencia [W] (b) J/J_c (c) Calor

Figura 3.11: Distribuciones en Julia sobre GPU para la Geometría (144,2) con Puntos Variables de Integración según región GPU I.



(a) Potencia [W] (b) J/J_c (c) Calor

Figura 3.12: Distribuciones en Julia sobre GPU para la Geometría (144,2) con 4000 Puntos Fijos de Integración para todas las regiones GPU II.

3.3. Modelo III: Incremento eficiencia en GPU

La evolución de la función `CreationVectorPotentialMatrix12PancakeCoils_CUDA` a la versión final del Modelo III, tal como se presenta en el algoritmo 7, se basa en un conjunto de modificaciones orientadas a mejorar la eficiencia computacional y la robustez numérica de los cálculos, sin alterar sustancialmente el cómputo de matrices, implementado exitosamente en la GPU en el Modelo II y que como demuestra el Cuadro 3.4 ya es notablemente eficiente, aunque esta depende del número de puntos de integración empleados.

En primer lugar, las matrices de salida K, MBr y MBz se inicializan con unas dimensiones de N filas por $2N$ columnas, a diferencia de la función original donde las matrices se definían como $2N$x$2N$. Esta estructura rectangular simplifica la resolución del problema al evitar el cálculo de la mitad inferior de las matrices K, MBr y MBz. Tras una exploración exhaustiva de la función $ApplySymmetry12PancakeCoils$ se descubrió que esta sección de las matrices no era empleada para ningún fin, siendo estos valores completamente descartados. Consecuentemente, esta reestructuración reduce la carga de trabajo al prevenir el cálculo innecesario de Nx$2N$ valores, lo cual supone una reducción de la mitad de cálculos necesarios respecto a las implementaciones anteriores.

La siguiente gran diferencia se encuentra en la introducción de **pre-cálculos de términos recurrentes** tales como `ri_sq = ri_val^2`, `dzi_cj = zi_val - cj_val`, `dzi_dj = zi_val - dj_val`, y el uso consistente de `cos_theta = cos(\theta)` y `sin_theta = sin(\theta)`. Esta técnica es muy efectiva en la optimización de código para GPUs, especialmente en un modelo donde las matrices `geometry.r_middle`, `geometry.r_low`, `geometry.r_high`, `geometry.z_middle`, `geometry.z_low` y `geometry.z_high` gozan de una gran simetría en la distribución de sus valores. La arquitectura interna de las matrices `geometry.r` contiene todos los elementos dentro de una misma columna idénticos, variando sus valores entre columnas. Alcanzada la mitad de las columnas dichos valores se repiten en el mismo orden. Las matrices `geometry.z`, sin embargo, exhiben un comportamiento opuesto. Cada columna contiene valores únicos internamente, que se repiten a lo largo de la primera mitad de columnas de la matriz. A partir de dicho punto las columnas restantes invierten el orden de los valores dentro de las columnas y niegan su valor. Esta redundancia en los valores se esquematiza en la Figura 3.13.

Para evitar recalcular estos valores repetidamente en cada operación dentro de los bucles de integración para cada hilo y en cada paso de la integración, se opta por su reutilización constante. Por ejemplo, `ri_val^2` o `ri_sq` aparece tanto en el numerador como en el denominador de `term1`, `term2` y `term3`, al igual que las diferencias `zi[idx] - cj[idy]`, por ejemplo. Al pre-calcular estos valores una sola vez al inicio del hilo y almacenarlos en variables temporales, se minimiza

$$\mathbf{geometry.r} = \begin{pmatrix} a & \dots & x & a & \dots & x \\ a & \dots & x & a & \dots & x \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ a & \dots & x & a & \dots & x \end{pmatrix}$$

(a) Estructura de `geometry.r`.

$$\mathbf{geometry.z} = \begin{pmatrix} v_1 & \dots & v_1 & -v_{n+1} & \dots & -v_{n+1} \\ v_2 & \dots & v_2 & -v_n & \dots & -v_n \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ v_n & \dots & v_n & -v_2 & \dots & -v_2 \end{pmatrix}$$

(b) Estructura de `geometry.z`.

Figura 3.13: Esquemas de las matrices de geometría, mostrando sus patrones de repetitividad.

considerablemente la redundancia computacional. En el contexto de las GPUs, donde miles de hilos ejecutan las mismas operaciones de manera multitudinaria en paralelo, esta reducción de operaciones por hilo se traduce en una disminución exponencial del tiempo de ejecución total, ya que cada ciclo de reloj ahorrado se multiplica por el número de hilos activos. De esta forma, este enfoque aprovecha la arquitectura SIMD de las GPUs, donde la uniformidad de las operaciones permite que el pre-cálculo beneficie a todos los elementos procesados en paralelo.

Finalmente, otro aspecto crucial es la **robustez mejorada de los cálculos de raíz cuadrada y de logaritmo** a través de expresiones como `sqrt(max(radX_sq, epsilon))` y `log(max(min(safe_num / safe_den, 1.0e30), 1.0e-30))`, que otorgan estabilidad numérica a las operaciones. En el dominio de los cálculos de punto flotante, los argumentos de las raíces y de los logaritmos pueden, debido a la propagación de errores de redondeo o pequeñas ineficiencias en los cálculos, volverse negativos o extremadamente pequeños. La implementación de estos mecanismos garantiza que los argumentos sean siempre un número positivo mínimo (`epsilon`, por ejemplo, $1{,}0e-10$), previniendo así la aparición de `NaNs` o `Infs` y asegurando que el cálculo permanezca numéricamente estable y válido. Estas optimizaciones en conjunto contribuyen a transicionar de un prototipo funcional a una implementación de alto rendimiento confiable y precisa para problemas de gran escala, cuyas mejoras se evaluarán en la siguiente sección.

Algorithm 7 Diferencias clave en el Algoritmo de Cálculo de Matrices (Julia con CUDA)

Function: CreationVectorPotentialMatrix12PancakeCoils_CUDA(*geometry*)
▷ Definición expandida de N y dimensiones de matriz de salida
$Nz \leftarrow geometry.Nz$, $Nr \leftarrow geometry.Nr$, $Nc \leftarrow geometry.Nc$, $Ne \leftarrow geometry.Ne$
$N \leftarrow Ne \cdot Nz \cdot Nr \cdot Nc$
$K \leftarrow$ CUDA.zeros(Float64, $N, 2 \cdot N$) ▷ Dims K salida: Filas N, Columnas $2N$
$MBr \leftarrow$ CUDA.zeros(Float64, $N, 2 \cdot N$)
$MBz \leftarrow$ CUDA.zeros(Float64, $N, 2 \cdot N$)
▷ Optimización en la transferencia de datos para ri
$ri \leftarrow$ CuArray(reshape($geometry.r\_middle, :, 1$))
$zi \leftarrow$ CuArray(reshape($geometry.z\_middle, :, 1$))
... ▷ Otras transferencias aj, bj, cj, dj como antes
▷ Ajuste en el número de puntos de integración
$n\_points\_low \leftarrow 1500$
$n\_points\_high\_K \leftarrow 2000$
$n\_points\_high\_M \leftarrow 3000$
▷ Definición dinámica de regiones para puntos de integración alta
$region1\_start \leftarrow 1$
$region1\_end \leftarrow N/2$
$region2\_start \leftarrow N/2 + 1$
$region2\_end \leftarrow N$
▷ Configuración de bloques para un grid rectangular
$blocks\_per\_grid\_x \leftarrow \lceil N/threads\_dim \rceil$
$blocks\_per\_grid\_y \leftarrow \lceil (2 \cdot N)/threads\_dim \rceil$
$blocks\_per\_grid \leftarrow (blocks\_per\_grid\_x, blocks\_per\_grid\_y)$
▷ Dentro del **cuda_kernel!**:
    ▷ Uso de ri con indexación dinámica
    $ri\_val \leftarrow ri[idx]$
    ▷ Mayor pre-cálculo de términos para eficiencia
    $ri\_sq \leftarrow ri\_val^2$
    $dzi\_cj \leftarrow zi\_val - cj\_val$
    $dzi\_dj \leftarrow zi\_val - dj\_val$
    $cos\_theta \leftarrow \cos(\theta)$, $sin\_theta \leftarrow \sin(\theta)$
    ▷ Robustez mejorada en raíces cuadradas para evitar NaNs/Infs
    $rad1a\_sq \leftarrow dzi\_dj^2 + aj\_val^2 + ri\_sq - 2{,}0 \cdot aj\_val \cdot ri\_val \cdot sin\_theta$
    $rad1a \leftarrow$ sqrt(max($rad1a\_sq, epsilon$))
for $i = 1$ to N **do**
    $K_{cpu}[i, i] \leftarrow$ hcubature(...)[1]
    return $K_{cpu}, MBr_{cpu}, MBz_{cpu}$

3.3.1. Evaluación de Resultados

Para el Modelo III no se va a incluir la tabla comparativa respecto a las características del Modelo II, pues intrísicamente se trata del mismo modelo excepto por la reducción de las dimensiones de las matrices de salida K, MBr y MBz, y por la inclusión de pre-cálculos de términos recurrentes en el cálculo e integración de dichas matrices. De hecho, como único apunte a este respecto se resalta que la complejidad algorítmica $O(N^2)$ de los Modelos I y II en Julia independientemente del uso de CPU o de GPU se sigue manteniendo, aunque ligeramente mejorado, pues los modelos GPU I'y GPU II'muestran un factor de crecimiento del tiempo de computación de 13.24x y de 13.01x para la geometría (40,4) y de 43.96x y 39.02x respectivamente.

Lo más destacable de este modelo reside en que muestra un comportamiento opuesto al observado en el Modelo II. En este caso, como indica el Cuadro 3.5, el escenario donde se utiliza una mayor cantidad de puntos de integración uniforme para todo el área, en concreto 4000 puntos, obtiene peores resultados y rendimientos para el cómputo de K y M que el modelo GPU I', que utiliza un número de puntos variable, 2000 puntos para todo el área de la matriz K y para las posiciones con poco error de la matriz M, y 4000 puntos para las posiciones con elevado error de M. La diferencia en Q para la geometría (20,2) es prácticamente despreciable, pero en el caso (40,4) la potencia disipada estimada por el modelo GPU Iés prácticamente coincidente con el valor de CPU 2171.00 W, mientras que el valor de GPU II'diverge ligeramente. Para (144,2) esta diferencia es aún más pronunciada tanto para GPU I'como para GPU II', si bien la estimación de GPU Iés más cercana. Para este modelo, por lo tanto, parece no ser necesaria la toma de decisión entre una mayor precisión o una mejor eficiencia, pues para números de puntos razonables para el modelo, valores menos elevados parecen inferir más rápidamente, como es lógico, pero sobre todo más precisamente el comportamiento del sistema SMES. No obstante, es importante recalcar la ganancia en eficiencia del cómputo de K y M se ve contrarrestado por la velocidad de convergencia de los solvers, dando lugar a unos tiempos totales prácticamente coincidentes entre GPU I'y GPU II'para las tres geometrías, aunque inferiores a los del Modelo II.

Respecto a los gráficos de potencia, densidad de corriente y calor disipado mostrados en las Figuras 3.14 a 3.19, cabe puntualizar los mismos detalles comentados para los modelos anteriores, donde la única diferencia aparente es perceptible en la visualización del calor disipado. A pesar del hallazgo realizado sobre la mayor precisión de los modelos GPU I'con menos puntos de integración, en ninguna de las visualizaciones se vislumbra un comportamiento diferente, lo cual indica que a alto nivel no existe una sustancial diferencia entre variar los puntos de integración en rangos, pues visualmente no se aprecian las diferencias.

Cuadro 3.5: Tiempos de Ejecución para Diferentes Geometrías en Julia con Modelo base y mejorado en GPU.

(a) Geometría (20,2)

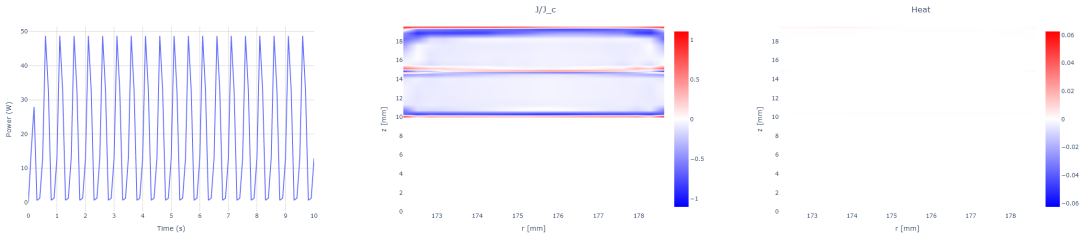
Característica	**GPU I**	**GPU I'**	**Δ [%]**	**GPU II**	**GPU II'**	**Δ [%]**
Mesh [s]	0.00158	0.0437	-2665.82	0.000393	0.005	-1172.26
Geometry [s]	0.0000013	0.000001	23.08	0.000001	0.0000001	90.00
K & M Matrix [s]	55.23	11.59	79.02	99.42	22.84	77.04
ϕ Matrix [s]	0.00205	0.00058	71.71	0.00558	0.000786	85.91
Symmetry [s]	0.0034	0.0171	-402.94	0.0044	0.05029	-1042.95
Solver [s]	106.22	132.70	-24.93	73.70	111.98	-51.94
Q [W]	185.50	188.07	–	185.45	189.46	–

(b) Geometría (40,4)

Característica	**GPU I**	**GPU I '**	**Δ [%]**	**GPU II**	**GPU II '**	**Δ [%]**
Mesh [s]	0.0071	0.0085	-19.72	0.0098	0.0098	0.00
Geometry [s]	0.000001	0.000001	0.00	0.000001	0.000001	0.00
K & M Matrix [s]	875.34	153.56	82.46	1613.09	297.15	81.58
ϕ Matrix [s]	0.0081	0.0044	45.68	0.0717	0.0055	92.33
Symmetry [s]	0.8084	0.7299	9.71	0.8395	0.1072	87.23
Solver [s]	3388.68	3728.03	-10.01	2459.63	2519.03	-2.42
Q [W]	2069.52	2169.52	–	2117.89	2211.06	–

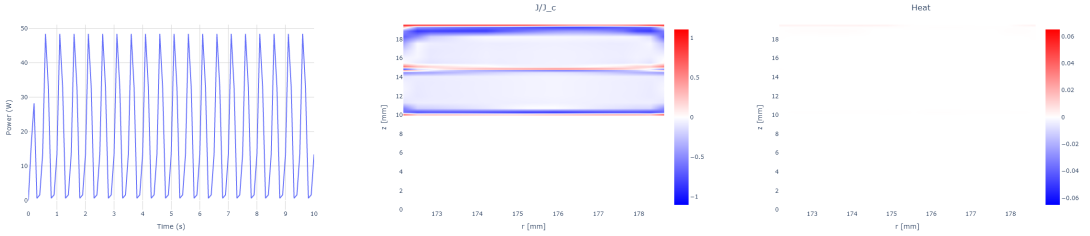
(c) Geometría (144,2)

Característica	**GPU I**	**GPU I '**	**Δ [%]**	**GPU II**	**GPU II '**	**Δ [%]**
Mesh [s]	0.0156	0.0201	-28.85	0.0173	0.0235	-35.83
Geometry [s]	0.000001	0.000001	0.00	0.000001	0.000001	0.00
K & M Matrix [s]	2779.77	509.54	81.67	5140.93	891.37	82.66
ϕ Matrix [s]	0.0157	0.0186	-18.47	0.0151	0.0187	-19.25
Symmetry [s]	2.5917	2.5435	1.86	25.6117	0.5103	98.01
Solver [s]	16624.24	18698.05	77.59	11895.14	14820.14	-6.39
Q [W]	12032.49	12759.48	–	12135.48	13157.18	–



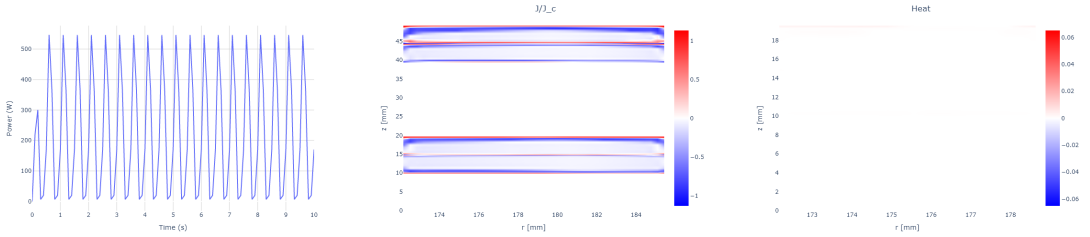
(a) Potencia [W] (b) J/J_c (c) Calor

Figura 3.14: Distribuciones en Julia sobre GPU optimizada para la Geometría (20,2) con Puntos Variables de Integración según región GPU I'.



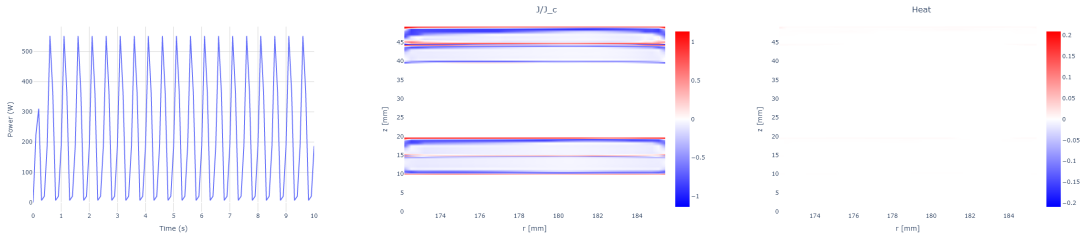
(a) Potencia [W] (b) J/J_c (c) Calor

Figura 3.15: Distribuciones en Julia sobre GPU optimizada para la Geometría (20,2) con 4000 Puntos Fijos de Integración para todas las regiones GPU II'.



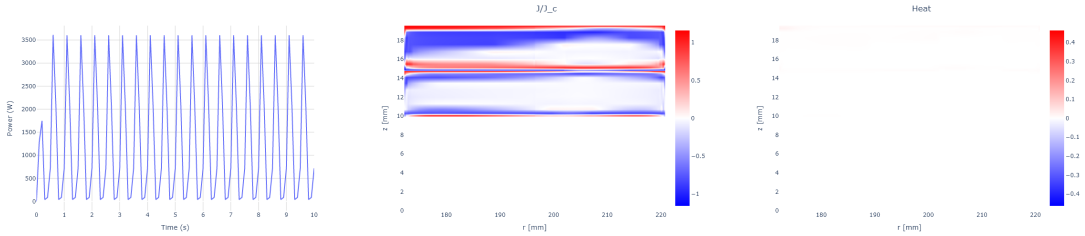
(a) Potencia [W] (b) J/J_c (c) Calor

Figura 3.16: Distribuciones en Julia sobre GPU optimizada para la Geometría (40,4) con Puntos Variables de Integración según región GPU I'.



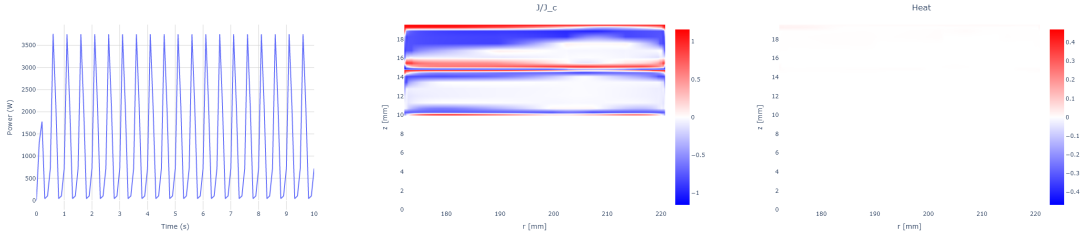
(a) Potencia [W] (b) J/J_c (c) Calor

Figura 3.17: Distribuciones en Julia sobre GPU optimizada para la Geometría (40,4) con 4000 Puntos Fijos de Integración para todas las regiones GPU II'.



(a) Potencia [W] (b) J/J_c (c) Calor

Figura 3.18: Distribuciones en Julia sobre GPU optimizada para la Geometría (144,2) con Puntos Variables de Integración según región GPU I'.



(a) Potencia [W] (b) J/J_c (c) Calor

Figura 3.19: Distribuciones en Julia sobre GPU optimizada para la Geometría (144,2) con 4000 Puntos Fijos de Integración para todas las regiones GPU II'.

3.4. Comparativa del Cómputo para las Matrices K y M

Para poder establecer una comparativa global y comparar el rendimiento incremental de todos los modelos conjuntamente, se ilustran en la Figura 3.20 el rendimiento de los cuatro modelos (MATLAB, Modelo I, Modelo II y Modelo III) para las diferentes geometrías. Puesto que para el modelado en GPU se ha utilizado un conjunto distinto de puntos de integración, estos se han incluido también para poder vislumbrar el comportamiento temporal cuando se realiza el balance entre precisión y tiempo. El caso GPU I se ha identificado con el sobrenombre "Puntos Var."dando a entender que no se ha utilizado un número homogéneo de puntos para realizar las integrales, mientras que para GPU II se ha indicado que se han empleado 4000 puntos en todo el área de integración. Para los modelos de MATLAB y de CPU en Julia el tiempo, independientemente del número de puntos, es el mismo para las geometrías.

Como se aprecia claramente en el gráfico, cada evolución del modelo en Julia supone un incremento en la eficiencia temporal. Los modelos Julia CPU y Julia GPU Modelo III suponen las dos aceleraciones más sustanciales, reduciendo los tiempos en comparación con MATLAB CPU en más de 6 y 12 órdenes de magnitud respectivamente. Este fenómeno se vuelve especialmente crítico para geometrías de mayor escala, donde los tiempos de cómputo sobre la CPU se vuelven extremadamente elevados. Esta mejora se traduce directamente en una **reducción drástica de los tiempos de simulación**, transformando lo que antes eran operaciones que podían tardar largar horas en MATLAB CPU, en procesos que se completan en cuestión de minutos o segundos en la GPU. El Modelo II, aunque mejora el rendimiento de la CPU de Julia supone el menor incremento en eficiencia comparativamente, si bien actúa como modelo puente para desarrollar el modelo final, el Modelo III de Julia sobre la GPU.

El Modelo III representa la implementación más optimizada, alcanzando una mejora superior al 95 % en el tiempo de cálculo respecto a MATLAB CPU para cualquier geometría y configuración. Esto significa que una tarea que en MATLAB tomaría aproximadamente 926.34 segundos (alrededor de 15 minutos), por ejemplo, el Modelo III la resuelve en tan solo 11.59 segundos. Incluso para configuraciones más exigentes como (40,4) 4000 Puntos o (144,2) con Puntos Variables, el Modelo III mantiene una impresionante mejora del 95.76 % y del 97.70 % sobre MATLAB CPU, reduciendo un proceso de 7007.5 segundos (más de 1.9 horas) a unos 297.15 segundos (aproximadamente 5 minutos) y otro de 22197.9 segundos (más de 6 horas) a apenas 509.54 segundos (aproximadamente 8 minutos).

Cuadro 3.6: Tiempos de Ejecución y Porcentaje de Mejora (MATLAB CPU vs. Julia GPU Modelo III)

Geometría y Configuración	**MATLAB CPU (s)**	**Modelo III (s)**	**Mejora (%)**
(20,2) Puntos Var.	926,34	11,59	98,75
(20,2) 4000 Puntos	926,34	22,84	97,53
(40,4) Puntos Var.	7007,50	153,56	97,81
(40,4) 4000 Puntos	7007,50	297,15	95,76
(144,2) Puntos Var.	22 197,9	509,54	97,70
(144,2) 4000 Puntos	22 197,9	891,37	95,98

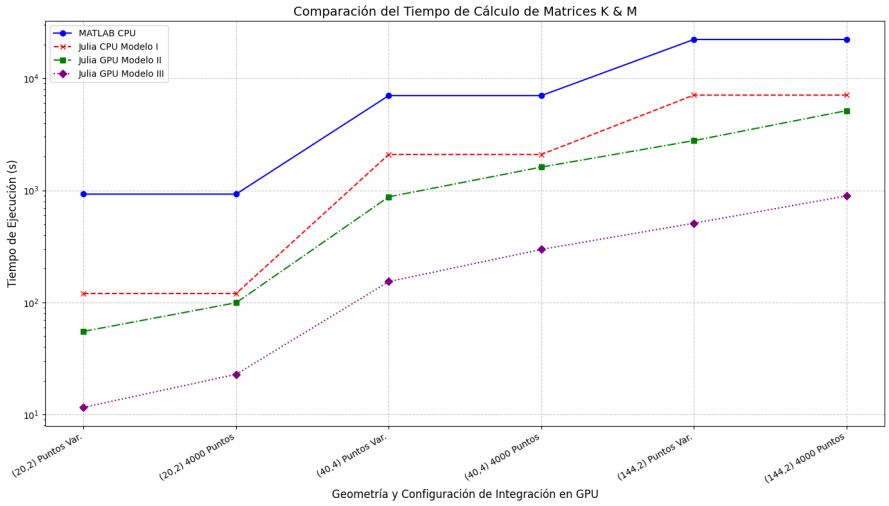


Figura 3.20: Comparación de los Tiempos de Ejecución de K y de M según la Geometría para los modelos en MATLAB y en Julia.

Capítulo 4

ODE Solvers

Para calcular la densidad de corriente, la potencia y el calor es necesario resolver ecuaciones diferenciales. Puesto que este cálculo, junto con la computación de las matrices K y M, representa uno de los elementos de mayor requerimiento computacional del sistema, merece una inspección más a fondo. La resolución de estas ODEs es crucial para simular el comportamiento dinámico del superconductor, permitiendo observar la evolución temporal de la corriente y sus magnitudes asociadas.

La principal ventaja que impulsa la migración a Julia es su teórico superior rendimiento, donde incluso los solvers de ODE más simples son más rápidos, demostrando mejoras de varios órdenes de magnitud en ecuaciones diferenciales, resolución no lineal y optimización [26]. Esta eficiencia se amplifica con la facilidad para la computación de alto rendimiento y el paralelismo, donde el ecosistema de Julia simplifica la aceleración por GPU con `CUDA` ofreciendo paralelismo multinúcleo y distribuido de forma integrada en muchas librerías, maximizando así el aprovechamiento del hardware.

La Figura 4.1 muestra una comparación de rendimiento de diversos solvers de Ecuaciones Diferenciales Ordinarias para el problema "Stiff 1: ROBER", un benchmark común para la eficiencia de integración numérica. El problema ROBER consiste en un sistema de tres ecuaciones diferenciales no lineales con constantes de tiempo muy diferentes, que se deriva de un modelo de reacción química. En el gráfico, el eje X representa el error y el eje Y indica el tiempo de ejecución en segundos en escala logarítmica. Se aprecia claramente una superioridad de los solvers de Julia, particularmente de las variantes "Julia: Rodas4 Static", "Julia: Rodas5 Static", "Julia: Rosenbrock23 Staticz "Julia: Rodas4". Estos solvers, representados con diferentes tonos de verde oscuro y claro, alcanzan niveles de error muy bajos con tiempos de ejecución significativamente menores en comparación con sus contrapartes de MATLAB (`ode23s` y `ode15s`, en naranja). Los solvers de MATLAB tienden a situarse en la parte superior del gráfico junto con otras

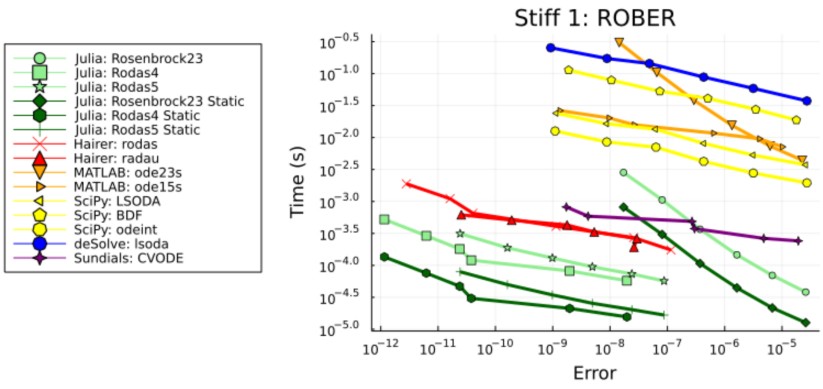


Figura 4.1: Comparación de distintos Solvers de ODEs. [26]

alternativas como SciPy y Sundials, indicando mayores tiempos de computación para un error dado. De esta manera los solvers de Julia destacan consistentemente como los más eficientes en este problema de ODE rígido.

4.1. Implementación ODEs

Considerando la mayor eficiencia de los solvers de Julia, particularmente de los métodos de la familia Rosenbrock, en comparación con implementaciones en otros lenguajes como MATLAB o SciPy, se han evaluado diversas variantes de estos solvers de Julia frente a ode15s, el solver de referencia en el modelo de MATLAB. La elección de los solvers se vio muy acotada debido al uso de la matriz Ks como matriz de masa, puesto que la mayoría de los métodos numéricos convencionales para ecuaciones diferenciales no están diseñados para su manejo. A continuación, se describen los solvers utilizados:

- **ode15s (MATLAB):** Este es un solver de orden variable para problemas de ecuaciones diferenciales ordinarias de tipo rígido, ampliamente utilizado en MATLAB. Emplea un método implícito de diferenciación hacia atrás y es conocido por su robustez y eficiencia en una amplia gama de problemas rígidos, adaptando dinámicamente el orden y el tamaño del paso para mantener la precisión y la estabilidad. Su uso es recomendado para resolver ecuaciones diferenciales algebraicas (EDAs) debido a su capacidad para manejar matrices de masa, incluso singulares (que caracteriza a las EDAs).

- **Rodas4 (Julia):** Se trata de un método de Rosenbrock de cuarto orden, clasificado como A-estable y rígidamente estable [27]. Su interpolador de tercer

orden "sensible a la rigidez"(stiff-aware interpolant) es crucial para mantener la precisión y la eficiencia en la estimación de soluciones intermedias. La propiedad de A-estabilidad garantiza la estabilidad para un amplio rango de tamaños de paso en problemas rígidos.

- **Rodas5P (Julia):** Constituye una evolución del método de Rosenbrock, siendo de quinto orden, también A-estable y rígidamente estable [27]. Su principal mejora radica en una estabilidad optimizada por la inclusión de "pasos de tiempo adaptativos"(adaptive time stepping embedding), lo que le permite manejar de forma más eficiente y precisa la selección del tamaño de paso en sistemas con rigidez variable o transiciones bruscas, ofreciendo mayor robustez y exactitud en problemas complejos.

- **ROS34PW2 (Julia):** Este solver es un método Rosenbrock-W de cuarto orden, específicamente diseñado para ser "rígidamente preciso"(stiffly accurate) en la resolución de sistemas de ecuaciones diferenciales algebraicas [27]. Su particularidad reside en su capacidad para manejar directamente las restricciones algebraicas de las EDAs, que surgen a raíz de la presencia de matrices de masa singulares o no invertibles.

Los solvers Rodas4 y Rodas5P fueron seleccionados conforme a las recomendaciones de la documentación oficial de Julia para la resolución de problemas caracterizados por matrices de masa constantes y la exigencia de alta precisión [27]. Por otro lado, ROS34PW2 fue incorporado para evaluar un método de la familia Rosenbrock-W, dado el rendimiento subóptimo observado con los solvers Rodas4 y Rodas5P, tal como se ilustra en la Figura 4.2.

La convergencia de los ODE solvers se ha testeado con los valores obtenidos tanto del modelo en MATLAB para ode15s como del Modelo I en Julia, debido a la equivalencia en los resultados. Puesto que se quiere realizar una comparativa ante condiciones idénticas para determinar qué solver es el más eficiente, los valores numéricos de los Modelos II y III no son seleccionables, pues presentan ligeras desviaciones de los valores de MATLAB. De esta manera, la elección de utilizar los resultados de la implementación sobre CPU en Julia garantiza la consistencia de los resultados.

A diferencia de lo que cabría esperar basándose en el rendimiento de los solvers para problemas de benchmark como el "Stiff 1: ROBER", los métodos Rodas5P y, aún más destacadamente, Rodas4, exhiben un rendimiento muy inferior al del solver ode15s. Esta disparidad se acentúa conforme aumenta la complejidad de la geometría. De hecho, para la geometría (144,2), el solver Rodas4 no logró la convergencia en un tiempo razonable, excediendo las 24 horas. Para la geometrías (40,4), por ejemplo, Rodas5P requirió más del doble de tiempo que ode15s, mientras que Rodas4 necesitó más de diez veces el tiempo del solver de MATLAB.

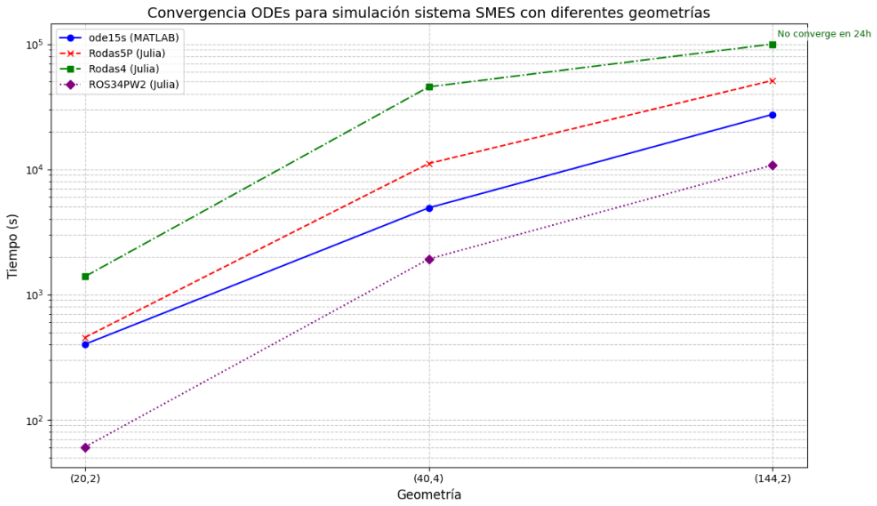


Figura 4.2: Comparación de la Convergencia de distintos Solvers de ODEs para las diferentes Geometrías.

No obstante, el método ROS34PW2 ofrece un rendimiento consistentemente mejorado en todas las geometrías evaluadas, desde la configuración más básica (20,2), donde consiguió una mejora en un factor de 6 en el tiempo de convergencia, hasta las geometrías más complejas (40,4) y (144,2), donde la mejora fue aproximadamente de 2.5x en ambos casos. Por consiguiente, ROS34PW2 se postula como un solver óptimo para lograr una convergencia eficiente en la resolución de las ODEs dentro de la simulación de sistemas SMES, con prestaciones notablemente superiores a ode15s, Rodas4 y Rodas5P.

Capítulo 5

Medidas Empíricas de Pérdidas AC

Para comprender la naturaleza de las pérdidas AC y la resistencia en las bobinas HTS, es fundamental analizar sus circuitos equivalentes, algunos de los cuales se ilustran en la Figura 5.1. Para bobinas HTS sin aislamiento, el circuito equivalente (Figura 5.1(a)) consiste en una inductancia de bobina (L_c) y dos resistencias (R_{sc} y R_f), donde R_{sc} representa la resistencia del superconductor y R_f es la resistencia de dirección radial, es decir la resistencia de contacto entre espiras. Cuando las bobinas HTS operan bajo corriente AC, R_{st} equivale a la resistencia estabilizadora de las bobinas HTS en dirección azimutal. En contraste, la resistencia para bobinas HTS con aislamiento es infinita en la dirección radial (R_f) [28], lo cual simplifica el flujo de corriente, al presentar resistencia únicamente en dirección azimutal. El circuito resultante se ilustra en la Figura 5.1(b).

Para simplificar el circuito de bobinas con aislamiento, R_{sc} y R_{st} se combinan en una única resistencia equivalente R_c utilizando la fórmula de resistencia en paralelo:

$$R_c = \frac{R_{sc} \times R_{st}}{R_{sc} + R_{st}} \tag{5.1}$$

El circuito simplificado de la bobina con aislamiento, que representa el modelo definitivo aplicado, se muestra en la Figura 5.1(c). Consecuentemente, la ecuación para obtener la tensión viene dada por:

$$u(t) = L_c \frac{di(t)}{dt} + R_c i(t) \tag{5.2}$$

donde $u(t)$ representa la tensión sobre los extremos de las bobinas HTS, e $i(t)$ representa la corriente que fluye a través de las bobinas HTS.

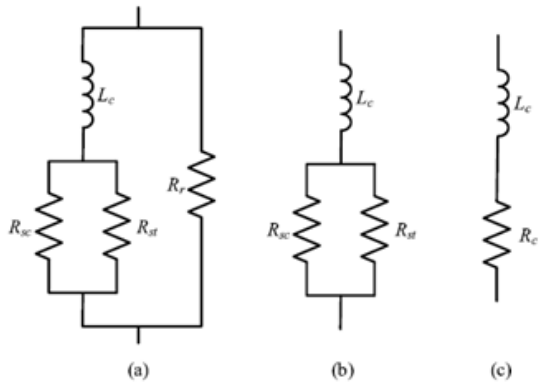


Figura 5.1: Circuitos equivalentes de bobinas HTS. (a) Bobina sin aislamiento. (b) Bobina con aislamiento. (c) Circuito equivalente simplificado de bobina con aislamiento. [28]

De acuerdo con la fórmula de Poynting, la potencia total transferida de una fuente a una carga puede ser interpretada de la siguiente manera:

$$\oint_S (\mathbf{E} \times \mathbf{H}) \cdot d\mathbf{S} = -\int_V (\mathbf{E} \cdot \mathbf{J}) dV - \frac{\partial}{\partial t} \int_V \left(\frac{\mathbf{D} \cdot \mathbf{E}}{2} + \frac{\mathbf{B} \cdot \mathbf{H}}{2} \right) dV \tag{5.3}$$

El primer término del lado derecho representa la potencia transformada en otras formas de energía, y el segundo término representa la derivada temporal de la energía almacenada en campos eléctricos y magnéticos. Aplicada a circuitos eléctricos, la ecuación 5.3 puede reescribirse como:

$$p(t) = v(t)i(t) = a(t) + r(t) \tag{5.4}$$

donde $p(t)$ es la potencia instantánea, $a(t)$ es la potencia activa instantánea y $r(t)$ es la potencia reactiva instantánea. $a(t)$ se calcula a partir de la resistencia R y la corriente $i(t)$, mientras que $r(t)$ se obtiene a partir de la inductancia L y la capacitancia C, como:

$$a(t) = Ri(t)^2 \tag{5.5}$$

$$r(t) = r_L(t) + r_C(t) = \frac{d}{dt} \left[\frac{1}{2} Li(t)^2 + \frac{1}{2} Cv_C(t)^2 \right] = p(t) - a(t) \tag{5.6}$$

donde $r_L(t)$ representa la derivada temporal de la energía almacenada en la inductancia, y $r_C(t)$ la derivada temporal de la energía almacenada en la capacitancia.

5.1. Métodos de Medida de Pérdidas AC

Tras identificar el origen de las pérdidas AC en los superconductores, a continuación se va a proceder a explicar los tres métodos de medida que existen para pérdidas AC empíricamente.

- El método magnético se basa en la medición del ciclo de histéresis como $e_{hy} = -\mu_0 \oint M(H_e)\, dH_e$ donde e_{hy} es la densidad de energía, μ_0 la permeabilidad magnética en el vacío, M la magnetización del material y H_e excitación magnética. La Figura 5.2 ilustra cómo la magnetización del material varía con la excitación magnética para diferentes permeabilidades relativas, formando un ciclo cerrado cuya área representa la pérdida de energía. Aunque se trate de un modelo con una alta sensibilidad, es sencillo que sufra degradaciones debido a interferencias ocasionadas por ruido electromagnético y sólo está disponible para muestras cortas [28].

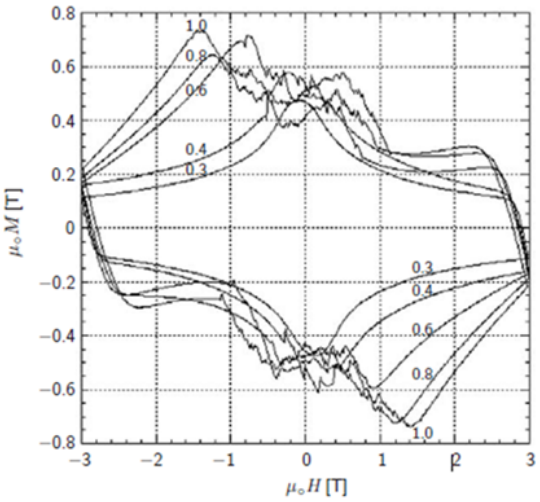


Figura 5.2: Ciclo de histéresis para diferentes valores de permeabilidad relativa [29].

- En el método calorimétrico la cinta o bobina son sumergidas en un baño de criógeno líquido, donde las pérdidas AC se deducen de la tasa de ebullición del criógeno que rodea la muestra a raíz del calor disipado. Otra manera de obtener las pérdidas AC es midiendo el incremento de la temperatura del equipamiento superconductor durante un período determinado.

- El método eléctrico, por su parte, consiste en aplicar una corriente variable en el tiempo (AC), y medir su caída de tensión. Dicha tensión se emplea para

calcular la potencia de calor disipada mediante la ecuación 5.7, que consiste obtiene la energía disipada en un ciclo y la divide por el ciclo de frecuencia:

$$P = \frac{1}{t_{cycle}} \int_0^{t_{cycle}} V(t) \cdot I(t)\, dt \tag{5.7}$$

donde $t_{cycle} = \frac{1}{f}$ es el tiempo que dura un ciclo, es decir, el periodo de la señal AC, $V(t)$ es la tensión medida entre los extremos del superconductor en función del tiempo e $I(t)$ la corriente alterna aplicada al circuito.

Para medir las pérdidas AC se ha seleccionado el método eléctrico, debido a la simplicidad de la realización del ensayo y de la toma de medidas. El diagrama 5.3 esquematiza el sistemas de medidas implementado para cuantificar las pérdidas AC empíricamente. El esquema se compone de una primera parte compuesta por una fuente de corriente que genera la señal senosoidal, el imán superconductor L, una resistencia en serie R y un diodo en paralelo a L y a R para proteger a la bobina HTS en caso de sobrevoltaje. La segunda parte se encarga de la toma de medidas. Para poder calcular la potencia de calor disipada mediante 5.7 es necesario obtener la corriente I por medio de un amperímetro, así como establecer la caída de tensión V en la bobina. Dichos valores son registrados en una tarjeta de extracción rápida con una tasa de 500000 muestras por segundos (500kS/s). Por último, la tarjeta se conecta a un ordenador para realizar el tratamiento y análisis de los datos.

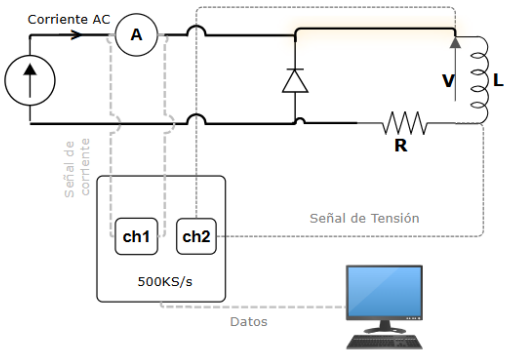


Figura 5.3: Esquema de Extracción de Medidas según el Método Eléctrico.

Las medidas experimentales de pérdidas AC se llevan a cabo simulando una corriente senoidal a frecuencia 1 Hz con amplitudes máximas de 25, 45, 65, 65 y 105A, y fijando como tensión mínima 5A para evitar el 0 en las fuentes de corriente, caracterizado por introducir una gran magnitud de ruido. Como se aprecia en la

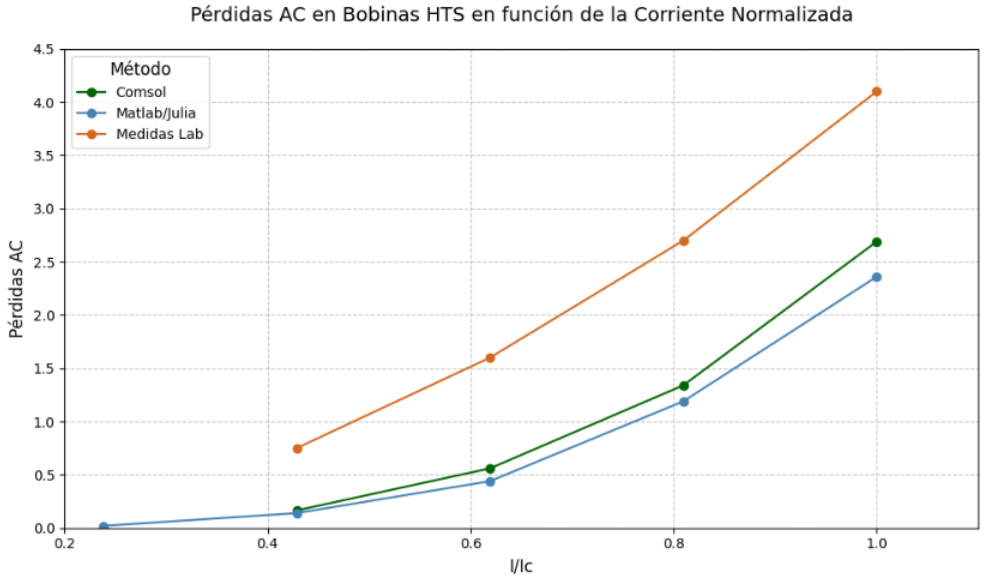


Figura 5.4: Comparativa de Pérdidas AC entre Medidas Empíricas y Modelos Teóricos por Software.

Figura 5.4 tanto el software de COMSOL Multiphysics como el modelo desarrollado ofrecen unos resultados muy similares de pérdidas AC confirmando la bondad de los modelos. Sin embargo, las medidas tomadas en el laboratorio, aunque consistentes con el comportamiento ligeramente exponencial mostrado por los modelos, parecen contener un error sustancial. El error contenido en las medidas, expresado mediante el desplazamiento de la curva hacia arriba, puede deberse a la susceptibilidad del entorno de medida, donde cualquier ligero error o ruido introducido por alguno de los elementos involucrados, sospechando principalmente del circuito de filtrado, puede estar ocasionando el desvío.

Capítulo 6

Conclusiones & Trabajo a Futuro

Para concluir el presente trabajo, se exponen en este capítulo los principales hallazgos y contribuciones, así como propuestas para futuras líneas de investigación.

Se ha construido de manera incremental un modelo computacional para simular el comportamiento de un sistema SMES en Julia. Inicialmente, el modelo operaba exclusivamente en CPU, sin embargo, la GPU fue identificada como una vía prometedora para acelerar sustancialmente la resolución de las matrices e integrales involucradas en el proceso. La eficiencia de los modelos desarrollados se evaluó sobre tres geometrías específicas: un doble pancake con 20 espiras (20,2), dos dobles pancakes compuestos por 40 espiras (40,4) y un doble pancake con 144 vueltas (144,2). Las mejoras implementadas en el modelo se han concentrado prioritariamente en el cómputo de las matrices K y M, las cuales, junto con la resolución de las ecuaciones diferenciales ordinarias, constituyen la práctica totalidad del tiempo de cómputo. Independientemente de la complejidad de la geometría, se ha logrado una mejora global en el tiempo de computación de las matrices superior al 95 % en comparación con el modelo base en MATLAB (desarrollado por J. Orea) y el modelo final en Julia, optimizado para GPU y denominado Modelo III.

Cabe destacar que, en el Modelo III, un número excesivamente elevado de puntos de integración puede resultar en mayores tiempos de cómputo sin una mejora proporcional en la precisión de los resultados. Esta observación sugiere la conveniencia de optar por un número inferior de puntos, o de implementar una distribución variable de estos puntos, concentrando un mayor número de puntos en las regiones donde el error es más significativo. Por ejemplo, para la geometría (40,4) con una distribución variable de los puntos, el Modelo III únicamente requiere 153.56 segundos ($\approx$ 2.56 minutos) frente a los 7007.50 segundos (más de 1.9 horas) requeridos por MATLAB, dando lugar a un diseño que permite modelar el comportamiento de un sistema SMES, incluso con geometrías complejas, en un lapso temporal significativamente reducido.

Además del cálculo de las matrices K y M, el segundo factor crítico en el rendimiento global del código es el tiempo de convergencia de los ODE solvers. El análisis exhaustivo de distintos solvers de Julia para diferentes geometrías ha demostrado que, si bien el solver ROS34PW2 presenta una velocidad de convergencia 2.5x mayor que la del solver de MATLAB, el cual ya exhibe una convergencia eficiente, otros solvers como Rodas5P requieren un tiempo excesivamente eleva-do, anulando las mejoras conseguidas en la eficiencia de la obtención de matrices. De esta manera, el cálculo optimizado en la GPU con la implementación de Ju-lia del Modelo III, en sinergia con el resolutor ROS34PW2, se consolidan como una opción altamente viable para la simulación en tiempo casi real de sistemas SMES independientemente de su complejidad y tamaño. La capacidad de obtener resultados en un marco temporal tan reducido es fundamental para combatir las demoras de cálculo que previamente suponían un significativo cuello de botella, logrando así alcanzar una simulación de sistema SMES sustancialmente más ágil.

6.1. Trabajo a Futuro

Como continuación del presente estudio, un elemento a investigar en profundidad para optimizar aún más la simulación de sistemas SMES sería la necesidad de emplear un número tan elevado de puntos de integración. El requerimiento actual de 2000 y 4000 puntos para garantizar la convergencia de los solvers parece excesivamente alto, pues una menor cantidad debería igualmente ser capaz de alcanzar valores precisos. Además, determinadas regiones de las matrices acaparan mayor tasa de error que otras, siendo de gran interés determinar el porqué de la distribución y concentración del error en determinadas regiones de integración.

Otra línea de interés en la búsqueda de mayores eficiencias computacionales comprende la implementación del modelo en clusters de GPUs. Si bien la optimización para una única GPU ha demostrado ganancias muy significativas, la simulación de problemas más complejos podría beneficiarse sustancialmente de la computación distribuida. Para ello se podría investigar y determinar si la escalabilidad del rendimiento se optimiza mejor mediante el aumento del número de núcleos de procesamiento en una única GPU, o si las mejores son mayores al distribuir la carga de trabajo entre múltiples GPUs. Este análisis permitiría identificar las arquitecturas de hardware más adecuadas y las estrategias de paralelización óptimas para problemas de gran envergadura en la simulación de sistemas SMES.

A nivel conceptual o de aplicabilidad del modelo, la capacidad de almacenamiento de energía magnética en un sistema SMES es una de sus características clave, que se ve exacerbada con la inclusión de materiales ferromagnéticos, como el hierro. En este contexto, un trabajo futuro de gran interés sería el estudio exhaustivo de la generación de pérdidas AC en bobinas superconductoras cuan-

do se incorpora un núcleo de hierro. Dicha investigación se enfocaría tanto en las pérdidas originadas en el material superconductor, como en aquellas inducidas dentro del propio núcleo ferromagnético, lo cual es crucial para obtener una caracterización completa del comportamiento energético del sistema.

Capítulo 7

Anexo

Los materiales superconductores permiten almacenar energía con una eficiencia prácticamente perfecta y sin disipación bajo condiciones operativas adecuadas. Esta singular propiedad de los superconductores, especialmente en el caso de los HTS, permitirá revolucionar la generación, conducción y almacenamiento de electricidad en el futuro, propiciando sistemas energéticos mucho más eficientes y resilientes. Por estos motivos se considera que la tesis se alinea directamente con los objetivos 7, 9, 11 y 13 de desarrollo sostenible establecidos por la Unión Europea.

El séptimo objetivo (Energía asequible y no contaminante), que persigue garantizar el acceso a una energía fiable, sostenible y moderna para todos, encuentra una fuerte sinergia con el decimotercer objetivo (Acción por el clima), enfocado en adoptar medidas urgentes para combatir el cambio climático y sus efectos. Desarrollos tecnológicos como los SMES para el almacenamiento son cruciales para la integración a gran escala de fuentes de energía renovables, como la eólica o la undimotriz, que por su naturaleza intermitente requieren soluciones de almacenamiento eficientes para garantizar la estabilidad de la red. Los SMES promueven no solo la generación de energía limpia al facilitar la penetración de renovables, sino tambien la innovación en el almacenamiento de energía contribuyendo al desarrollo de infraestructuras sostenibles.

En noveno objetivo (Industria, innovación e infraestructura), se promueve mediante la investigación y el desarrollo de metodologías computacionales avanzadas en el campo de los superconductores. Avanzar en la optimización del cálculo de pérdidas AC en sistemas SMES, tal como se aborda en este trabajo, impacta directamente en el diseño, la eficiencia y la viabilidad económica de soluciones energéticas innovadoras para el sector. La innovación en tecnologías de almacenamiento y generación de energía basadas en superconductividad es esencial para el desarrollo de infraestructuras energéticas más robustas, eficientes y sostenibles.

Finalmente, el undécimo objetivo (Ciudades y comunidades sostenibles), que busca lograr que las ciudades y los asentamientos humanos sean inclusivos, segu-

ros, resilientes y sostenibles, se ve directamente apoyado por la eficiencia que ofrecen los materiales superconductores. La implementación de sistemas SMES puede contribuir significativamente a una gestión más inteligente y eficaz de la demanda energética en entornos urbanos y comunitarios, estabilizando las redes eléctricas locales y reduciendo la probabilidad de interrupciones. La capacidad de proporcionar un suministro energético más confiable y de alta calidad es fundamental para asegurar el funcionamiento de las infraestructuras críticas en ciudades modernas, minimizando las pérdidas energéticas y sentando las bases para comunidades más sostenibles y energéticamente autónomas.

Bibliografía

[1] Mohammad Yazdani-Asrami et al. "Roadmap on artificial intelligence and big data techniques for superconductivity". En: *Superconductor Science and Technology* 36.4 (2023), pág. 043501. DOI: 10.1088/1361-6668/acbb34.

[2] Boyang Shen, Francesco Grilli y Timothy Coombs. "Overview of H -Formulation: A Versatile Tool for Modeling Electromagnetics in High-Temperature Superconductor Applications". En: *IEEE Access* PP (2020), págs. 1-1. DOI: 10.1109/ACCESS.2020.2996177.

[3] Xing Luo et al. "Overview of current development in electrical energy storage technologies and the application potential in power system operation". En: *Applied Energy* 137 (2015), págs. 511-536. DOI: https://doi.org/10.1016/j.apenergy.2014.09.081.

[4] Carlos Hernando et al. "Optimization of High Power SMES for Naval Applications". En: *IEEE Transactions on Applied Superconductivity* 33.5 (2023), págs. 1-5. DOI: 10.1109/TASC.2023.3250169.

[5] Hongye Zhang et al. "Alternating Current Loss of Superconductors Applied to Superconducting Electrical Machines". En: *Energies* 14.8 (2021). DOI: 10.3390/en14082234.

[6] Simon Otten y Francesco Grilli. "Simple and Fast Method for Computing Induced Currents in Superconductors Using Freely Available Solvers for Ordinary Differential Equations". En: *IEEE Transactions on Applied Superconductivity* 29.8 (2019), págs. 1-8. DOI: 10.1109/TASC.2019.2949240.

[7] Javier Orea Rufino. "Modelado y Validación de Pérdidas AC en Imán Superconductor de Alta Temperatura". Tesis de mtría. Universidad Pontificia Comillas, 2024.

[8] Anang Dadhich, Philippe Fazilleau y Enric Pardo. "A novel and fast electromagnetic and electrothermal software for quench analysis of high field magnets". En: *Superconductor Science and Technology* 37.9 (2024), pág. 095024. DOI: 10.1088/1361-6668/ad68d3.

[9] Enric Pardo y Francesco Grilli. "Electromagnetic Modeling of Superconductors". En: mar. de 2023, págs. 1-105. ISBN: 978-981-12-7143-4. DOI: 10.1142/9789811271441_0001.

[10] Soon-Gil Jung et al. "High critical current density and high-tolerance superconductivity in high-entropy alloy thin films". En: *Nature Communications* 13 (2022). DOI: 10.1038/s41467-022-30912-5.

[11] D K Hilton, A V Gavrilin y U P Trociewitz. "Practical fit functions for transport critical current versus field magnitude and angle data from (RE)BCO coated conductors at fixed low temperatures and in high magnetic fields". En: *Superconductor Science and Technology* 28.7 (2015), pág. 074002. DOI: 10.1088/0953-2048/28/7/074002.

[12] Felix Huber et al. "The T-A formulation: an efficient approach to model the macroscopic electromagnetic behaviour of HTS coated conductor applications". En: *Superconductor Science and Technology* 35 (mar. de 2022). DOI: 10.1088/1361-6668/ac5163.

[13] *MATLAB*. The MathWorks, Inc. Natick, Massachusetts, 2024. URL: https://www.mathworks.com.

[14] Cao Jingying et al. "Multi-physics coupling finite element analysis of 10kV tri-axial HTS cable". En: *E3S Web of Conferences* 118 (2019), pág. 02056. DOI: 10.1051/e3sconf/201911802056.

[15] Soumen Pal et al. "A next-generation dynamic programming language Julia: Its features and applications in biological science". En: *Journal of Advanced Research* 64 (2024), págs. 143-154. DOI: https://doi.org/10.1016/j.jare.2023.11.015.

[16] Jeff Bezanson et al. "Julia: A fresh approach to numerical computing". En: *SIAM Review* 59.1 (2017), págs. 65-98. DOI: 10.1137/141000671.

[17] The Julia Language. *Julia Micro-Benchmarks*. 2024. URL: https://julialang.org/benchmarks/.

[18] The Julia Language. *JIT Design and Implementation*. 2025. URL: https://docs.julialang.org/en/v1/devdocs/jit/#.

[19] JuliaGPU. *CUDA.jl: Julia wrapper for the CUDA library*. GitHub repository. URL: https://github.com/JuliaGPU/CUDA.jl.

[20] Tim Besard, Christophe Foket y Bjorn De Sutter. "Effective Extensible Programming: Unleashing Julia on GPUs". En: *IEEE Transactions on Parallel and Distributed Systems* (2018). DOI: 10.1109/TPDS.2018.2872064.

[21] University College London. *Computational Complexity*. https://github-pages.ucl.ac.uk/research-computing-with-cpp/07performance/sec01Complexity.html. University College London (UCL), 2025.

[22] Steven Reeves. *CPU vs GPU, and CUDA C/C++*. AMS 148 Chapter 2. Course material/Lecture notes, University of California, Santa Cruz. 2018. URL: https://ams148-spring18-01.courses.soe.ucsc.edu/system/files/attachments/note2_4.pdf.

[23] Jack Holland. *The Anatomy of a Computer: Part 1 of 4*. 2014. URL: https://medium.com/understanding-computer-science/the-anatomy-of-a-computer-part-1-of-4-4fda1928c774.

[24] An Li et al. "Implementation of a Fully-Parallel Turbo Decoder on a General-Purpose Graphics Processing Unit". En: *IEEE Access* 4 (2016), págs. 1-1. DOI: 10.1109/ACCESS.2016.2586309.

[25] Ebubekir Buber y Banu Diri. "Performance Analysis and CPU vs GPU Comparison for Deep Learning". En: *Computer Engineering Department Yildiz Technical University Istanbul* (2018), págs. 1-6. DOI: 10.1109/CEIT.2018.8751930.

[26] Rackauckas, Chris. *Getting Started with Julia's SciML for the MATLAB User*. https://docs.sciml.ai/Overview/dev/comparisons/matlab/. SciML, 2025.

[27] Christopher Rackauckas y Qing Nie. "DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia". En: *The Journal of Open Research Software* 5.1 (2017). DOI: 10.5334/jors.151.

[28] Xin Li et al. "A novel AC loss measurement method for HTS coils based on parameter identification". En: *Superconductor Science and Technology* 35 (2022), pág. 065021. DOI: 10.1088/1361-6668/ac6bc8.

[29] Kamil Etxagibel Begic. "Caracterización de una bobina superconductora de alta temperatura". Tesis de mtría. Universidad Carlos III Madrid, 2024.