TRABAJO FIN DE GRADO

Análisis Numérico de modelos regidos por Ecuaciones Diferenciales. Simulación e Implementación en Python.

Realizado por

Ana Casado Sánchez

Para la obtención del título de

Doble Grado en Ingeniería Informática - Tecnologías Informáticas y Matemáticas

Dirigido por

Dña. Macarena Gómez Mármol

Realizado en el departamento de

Ecuaciones Diferenciales y Análisis Numérico

Convocatoria de Junio, curso 2022/23

Agradecimientos

Quiero agradecer, en primer lugar, a mi directora Macarena por su exquisito trato y, sobre todo, por su paciencia. Sin su guía, este trabajo no podría haberse llevado a cabo.

Indiscutiblemente, debo de dar las gracias a mis padres, Juan y Patrocinio, que han permanecido a mi lado y me han apoyado durante esta etapa, una etapa que no ha resultado nada fácil debido a diversas circunstancias personales. A pesar de no haber dado la mejor versión de mí misma en algunas ocasiones, siempre he podido contarles todo con total confianza gracias al amor infinito que compartimos.

Además, quiero agradecer a Merche, a Alicia, al doctor Lozano y a su equipo el maravilloso trabajo que realizan y cómo me han ayudado a crecer como persona.

El otro gran pilar de mi vida durante estos años ha sido mi familia, no únicamente aquella forjada por lazos de sangre, sino también a la que me une el corazón. Todos me han hecho sentirme muy querida y han valorado el esfuerzo que he hecho. Quiero destacar a mi perrito Epsilon que hacía palmitas para animarme.

Por último, quiero mostrar mi gratitud a todos los amigos que he conocido en esta etapa. Su presencia diaria en la facultad hacían mucho más llevaderos mis días y espero haber alegrado también los suyos.

Resumen

La presente memoria está dedicada al estudio numérico de las ecuaciones diferenciales, tanto desde el enfoque teórico como desde el computacional, complementando las competencias adquiridas tanto en el grado en *Matemáticas* como en el grado en *Ingeniería Informática*. Consideramos el caso de sistemas diferenciales ordinarios y el de ecuaciones en derivadas parciales, que tratamos desde el punto de vista variacional. La resolución numérica efectiva de los problemas considerados, se lleva acabo usando Python gracias al módulo numpy, que es fundamental para la programación científica. En el caso de ecuaciones en derivadas parciales, nos apoyamos en la librería FEniCS y complementamos las representaciones gráficas utilizando ParaView.

Palabras clave: Python, Ecuaciones Diferenciales, Análisis Numérico, Elementos Finitos, Runge-Kutta

Abstract

This work focuses on the numerical analysis of differential equations both from the theoretical approach and from the computational one. In this way, all the knowledge learned in Math's degree and Computer Engineering's degree gets together and it can be expanded. We consider ordinary differential systems and partial differential equations, studied from the variational formulation's point of view. The numerical resolution of the problems considered is done by using Python thanks to the library numpy, which is the main one when programming scientific stuff. For partial differential equations, we use Python's module FEniCs and we represent the results using both Python and ParaView.

Keywords: Python, Differential Equations, Numerical Analysis, Finite Elements, Runge-Kutta

Índice general

1. Introducción

Con el nacimiento del Cálculo Diferencial en el siglo XVII, las ecuaciones diferenciales comenzaron a abrirse paso como herramientas fundamentales en la construcción de los modelos matemáticos. El siguiente gran hito histórico en este sentido fue el surgimiento del Análisis Funcional a principios del siglo XX de la mano de figuras como Hilbert y Banach, que permitieron abordar los problemas de forma más general como ecuaciones planteadas en espacios de funciones, que son infinitodimensionales. Destacamos en este sentido el nacimiento de la teoría de distribuciones de Schwartz, inspirado en resultados previos debido a Sóbolev.

Posteriormente, en la segunda mitad de siglo, el creciente desarrollo de los computadores llevó a los matemáticos a utilizarlos como herramienta de cálculo para poder resolver de forma aproximada modelos complejos basados en ecuaciones diferenciales. En la mayoría de los casos no es conocida una solución analítica.

Dado que las limitaciones del mundo físico nos impiden replicar el infinito matemático, el análisis numérico permite aproximar con un pequeño margen de error problemas infinitodimensionales por problemas finitodimensionales, los cuales pueden ser implementados en nuestras máquinas. Esto es justo lo que se pretende con este trabajo: la resolución numérica de modelos reales regidos por ecuaciones diferenciales utilizando Python, el lenguaje con mayores expectativas en la actualidad.

Debido a la gran cantidad de problemas prácticos modelados por ecuaciones diferenciales existentes en la actualidad, se trata de un tema de estudio muy versátil, con aplicaciones en múltiples campos, como Física, Ingeniería, Biología, Ciencias Sociales...

Asimismo, la realización de este trabajo complementa la formación académica de la alumna desde el punto de vista teórico y numérico, ampliando la base adquirida gracias a las diversas asignaturas del grado en *Matemáticas*.

Análogamente, el grado en *Ingeniería Informática*, ha preparado a la alumna para programar en diversos lenguajes y documentarse para adquirir la destreza necesaria para utilizar diversas herramientas software. Se ha escogido Python para la resolución de los modelos ya que, si bien es cierto que existen otras herramientas software orientadas a esta función, como Matlab, son comerciales, cerradas (utilizan muchos comandos con una estructura muy concreta que imposibilita la libertad completa a la hora de formular problemas) y poseen escasa integración con otras tecnologías.

Python, sin embargo, es Open Source, por lo que gran parte de la documentación oficial está al alcance de cualquier usuario. Asimismo, existen multitudinarios repositorios de Github y bibliotecas, tanto internas como externas, desarrollados y revisados por usuarios de todo el mundo a modo de comunidad. Por tanto, trabajar

con Python para la resolución numérica, permite aprender un enfoque de este lenguaje que no se trata en el grado. Además, obliga a trabajar con herramientas de visualización gráfica.

De esta forma, la elaboración de la presente memoria combina y complementa los conocimientos obtenidos en ambas titulaciones, sirviendo de colofón final del *Doble Grado en Ingeniería Informática-Tecnologías Informáticas + Matemáticas*.

La memoria está estructurada como sigue :

En el Capítulo 2, se presenta la iniciación y planificación del trabajo, ya que es una competencia de carácter fundamental del grado en *Ingeniería Informática* el poder organizar nuestro TFG como si de un proyecto se tratase. En este aspecto, durante la iniciación se realiza un estudio de la viabilidad del trabajo, se analizan los requisitos para poder fijar los objetivos y el alcance del mismo.

En cuanto a la planificación, se realiza una estimación de las tareas e hitos fundamentales y se distribuyen en el tiempo mediante plazos a modo de cronograma. Además, se proporcionan planes de contingencia para los diversos riesgos existentes en el desarrollo del TFG, se enumeran indicadores de calidad del trabajo y se planifica cómo será la comunicación entre la tutora y la alumna.

En el Capítulo 3, se estudia el análisis numérico de los problemas de Cauchy para sistemas diferenciales ordinarios. Para ello, comenzamos recordando algunos resultados teóricos fundamentales, entre los que destacamos la existencia y unicidad de solución maximal.

A continuación, se introducen los métodos de un paso y se definen varios conceptos fundamentales, destacando la consistencia, la estabilidad y la convergencia; esta última vendrá dada como consecuencia de las dos anteriores en la mayoría de casos. Asimismo, se define el orden de un método y, basándones en este, se realiza un estudio del error de discretización.

Finalmente, concluimos este capítulo introduciendo los métodos de Runge-Kutta y los tableros de Butcher de los métodos que incluiremos en la parte de resolución con Python. Considemos principalmente el caso de métodos encajados donde el paso es variable, proporcionando una resolución más precisa.

El Capítulo 4, concluye la parte de sistemas diferenciales ordinarios. Se implementa con Python de forma general el algoritmo correspondiente a un método de Runge-Kutta encajado a partir de su tablero de Butcher. Para ello, indicamos la configuración del entorno necesaria (que se traduce en la inclusión de matplotlib y numpy, que es la combinación más importante en cuanto a programación científica se refiere) y exponemos el código realizado debidamente organizado y explicado. Asimismo, se almacenan los tableros de los métodos que vamos a utilizar para los tests. En particular, se realizan tests numéricos de validación para los problemas del péndulo y de

la peonza, que son en 2D y 3D, respectivamente, lo que permite realizar representaciones gráficas.

Los capítulos 5 y 6 se dedican al estudio de problemas de ecuaciones en derivadas parciales. Al igual que con los problemas de Cauchy, el primero de estos capítulos será puramente teórico y el siguiente práctico.

En el Capítulo 5, comenzamos exponiendo unos complementos básicos de Análisis Funcional donde se introducen los espacios en los cuales se buscan las soluciones, así como sus propiedades principales. A continuación, se define qué vamos a entender por solución de un problema lineal elíptico a través de su formulación variacional. Destacamos el teorema de Lax-Milgram para la existencia y unicidad de solución de estos problemas.

Desde el punto de vista numérico, introducimos la aproximación de Galerkin y, como caso particular de esta, se estudian los elementos finitos de tipo Lagrange, destacando los elementos finitos simpliciales y los paralelotopos. De hecho, los simpliciales son los que utilizaremos en los programas. Los elementos finitos permiten trabajar con dominios muy generales y en su resolución intervienen matrices huecas, que son fáciles de tratar desde el punto de vista computacional y requieren menor coste de memoria. Realizamos también un estudio del error de estos métodos aplicado a la resolución de problemas elípticos de segundo orden.

En el capítulo 6, resolvemos diversos problemas de EDPs utilizando Python. En primer lugar, estudiamos la propagación de una onda sobre una membrana elástica cuadrada fijada en los bordes. Gracias a la sencillez de la geometría, no es difícil construir una triangulación. Debido a esto, se programa el método de los elementos finitos en Python sin utilizar ninguna librería dedicada a esto. En la memoria comentamos cómo se llevan a cabo estos cálculos y aprovechamos para introducir la formulación variacional a este tipo de problemas. Hay que tener en cuenta que es un problema evolutivo cuya formulación variacional no corresponde a lo tratado en el capítulo 5.

Para el resto de problemas resueltos, se usa la librería FEniCS. Dicha librería está construida para equipos con sistema operativo Linux. Por este motivo, se expone cómo configurar el entorno de programación para poder trabajar con FEniCS en cualquier equipo evitando las posibles incompatibilidades entre el sistema operativo y los elementos software utilizados.

Comenzamos tratando un problema muy simple, la ecuación de Poisson con segundo miembro constante en un disco, para validar los resultados del capítulo 5. Posteriormente, mostramos cómo calcular el primer autovalor del Laplaciano en una bola utilizando el método de la potencia normalizado.

A continuación, se tratan los dos ejemplos principales de este apartado. El primero se refiere a la deformación de una placa tridimensional elástica sujeta por un pilar

en el centro y sometida a la fuerza de la gravedad. En el segundo ejemplo estudiamos la difusión del calor entre dos entornos conectados por una interfaz, tanto de forma estacionaria como evolutiva. En todos los problemas resueltos en este sección se utiliza tanto Python como el software ParaView para realizar representaciones gráficas.

2. Iniciación y Planificación

En este capítulo se desarrollan los procesos referentes a las Fases de Iniciación y Planificación de este TFG, satisfaciendo así la competencia básica de gestión de proyectos que todo TFG del grado en Ingeniería Informática-Tecnologías Informáticas ha de cumplir.

2.1. Iniciación

Comenzamos tratando la Fase de Iniciación. Para esto, en esta sección se estudia la viabilidad del trabajo a realizar, así como se efectúa un análisis de requisitos para establecer, en líneas generales, máximas que han de seguirse para la correcta realización de dicho trabajo.

2.1.1. Estudio de la viabilidad

Desde el punto de vista técnico, llevar a cabo este TFG es posible, dado que los conocimientos obtenidos durante los estudios realizados, dotan de la base necesaria para poder estudiar los conceptos matemáticos que se van a tratar, así como la base para programar utilizando el lenguaje Python.

Asimismo, el material bibliográfico necesario es accesible sin costes económicos, gracias a que los libros utilizados están en la biblioteca de la Facultad de Matemáticas. Para la parte informática, el lenguaje de programación Python es Open Source y, por tanto, la documentación oficial de las librerías más relevantes están a libre disposición de los usuarios. Asimismo, el software FEniCS y la documentación referente al mismo también es accesible de forma gratuita gracias a repositorios de Github y la web de la API.

Por otra parte, no se predicen riesgos de gran envergadura:

- Eventuales retrasos por la imposibilidad de compaginar la realización del TFG con otras asignaturas y actividades.
- Dificultades para comprender algunas de los métodos numéricos a utilizar.
- Problemas implementando alguno de los algoritmos.

Ninguna de estas circunstancias se traduciría en graves pérdidas, por lo que no impide la realización efectiva del trabajo.

2.1.2. Análisis de requisitos

En esta sección definimos con mayor concreción qué (y hasta dónde) se va a hacer en este trabajo. En nuestro caso, el objetivo final puede definirse como el estudio de diversos métodos de resolución de ecuaciones diferenciales desde el punto de vista del Análisis Numérico.

Asimismo, este trabajo tendrá una fuerte componente informática, que consistirá en la programación de los métodos estudiados, así como el estudio de bibliotecas específicas dedicadas a este tipo de problemas. Todo este desarrollo se hará en el lenguaje Python. Este trabajo se complementará con el estudio y simulación con la programación desarrollada, de situaciones de interés en distintos campos de la Ciencia.

Al ser un TFG con un alto contenido matemático, se consideran los siguientes requisitos formales propios de cualquier trabajo propio de esta ciencia:

- Uso correcto del lenguaje matemático.
- Indicación explícita del marco de hipótesis en el que nos situamos, así como de la notación que se va a utilizar para conseguir la claridad formal.
- Inclusión de proposiciones, teoremas, corolarios y observaciones acompañados de números para referenciarlos correctamente.
- Realización de demostraciones para justificar con rigor las afirmaciones aportadas.

Sobre el análisis numérico y la resolución de los modelos:

- Uso de técnicas como mallado de superficies.
- Combinación de métodos numéricos de gran importancia, tales como Runge-Kutta, descenso por el gradiente y elementos finitos de Lagrange.
- Estudio de convergencia y error de las aproximaciones encontradas.

Sobre la parte correspondiente a la implementación de los métodos en Python será necesario:

- Coherencia del nombre otorgado a los parámetros y variables en el código con los utilizados en la parte teórica.
- Obtención de aproximaciones con un margen de error muy escaso.
- Eficiencia computacional en los algoritmos implementados.
- Realización de simulaciones gráficas de los resultados obtenidos, para poder entenderlos con mayor claridad gracias a la presencia del componente visual. Dependiendo de las características propias del modelo en cuestión, se optará por gráficos en dimensión 2 o 3. Será interesante recopilar imágenes llamativas para incluirlas en el documento referente al análisis numérico, al mismo tiempo

que se realicen vídeos que serán mostrados en la defensa del TFG y cuyo código se dejará propuesto al lector interesado para su posterior ejecución.

A continuación, debe fijarse el alcance del proyecto, especificando de forma precisa qué se va a hacer y qué no se va a realizar en este TFG.

- No se trata de un trabajo orientado a indagar sobre la librería gráfica de Python matplotlib, pero las funcionalidades referentes a colores, escalas, leyendas, títulos y tamaño deberán tenerse en cuenta para poder realizar los gráficos y animaciones de la forma más adecuada posible para visualizar los resultados de la forma deseada.
- La memoria deberá constar de un abstract, un índice, una introducción, dos capítulos puramente matemáticos y dos capítulos de resolución de problemas con Python haciendo uso de la teoría matemática descrita. Finalmente deberán aportarse las referencias utilizadas. Opcionalmente, la alumna podrá incluir agradecimientos entre el abstract y el índice si así lo desea.
- La redacción será clara, concreta y concisa.
- Las páginas estarán numeradas y presentarán una estructura de encabezamiento, pie de página y justificación de margen a ambos lados uniforme.
- Se utilizarán números a modo de índices no solo en los elementos formales matemáticos, sino también en gráficos o expresiones relevantes. Dichos índices deberán poseer enlaces que permitan acceder al contenido que referencian, facilitando al lector su labor.
- Se respetarán las reglas ortográficas propias del castellano en la medida en que estas no dificulten el desarrollo matemático del contenido.

Por último, este TFG está valorado en 18 créditos, lo que se traduce en una estimación de 450 horas de trabajo por parte de la alumna.

2.2. Planificacion

En este apartado hay que concretar las actividades a realizar, y de entre ellas los hitos más destacables, que generalmente irán acompañados de algún entregable. Asimismo, se deben incluir diversos planes de gestión: de riesgos, de calidad, de recursos humanos, de adquisiciones, de interesados, de comunicaciones.

2.2.1. Análisis temporal

En esta sección vamos a realizar la identificación de las actividades, tareas e hitos, así como se determinarán los plazos para alcanzar cada uno de los hitos propuestos.

El primer hito relevante es la entrega del acuerdo entre alumna y tutora tras algunas reuniones para concretar la propuesta a presentar y organizar el trabajo a realizar. Si bien es cierto que para los alumnos de la doble titulación con informática no se estableció una fecha límite de forma excepcional al ser la primera promoción que realiza el TFG, se acordó completar este hito al inicio de noviembre.

Dado que la alumna tenía una carga de 36 créditos (aparte del TFG) durante el primer cuatrimestre, se decidió que durante este período solo se realizarían labores de documentación en la medida de lo posible, comenzando con el desarrollo del trabajo en febrero.

Para los capítulos de análisis numérico, las tareas involucradas son:

- Leer material bibliográfico relacionado con la materia a desarrollar.
- Redactar con formalismo matemático (estructura basada en proposiciones, teoremas, demostraciones, etc) los fundamentos teóricos aprendidos y organizarla en secciones.

Para los capítulos de implementación y resolución con Python, las tareas involucradas son:

- Leer material bibliográfico relacionado con los módulos a utilizar.
- Idear algoritmos a modo de pseudocódigo previos a su implementación.
- Preparar el entorno de trabajo con los módulos y software necesario.
- Solventar errores e incompatibilidades entre las distintas tecnologías que intervienen.
- Realizar tests de validación para los métodos numéricos.
- Generar gráficos y animaciones tanto en 2D como en 3D, documentándose previamente para ello.

De este modo, los hitos previos a la fecha límite para la entrega del documento en la primera convocatoria (13 de junio de 2023), son la entrega de los distintos capítulos, así como la inclusión de las correcciones realizadas por la tutora. El cronograma resultante para los hitos correspondientes a los distintos capítulos es el siguiente:

Hito	Fecha límite
Entrega del capítulo Análisis Numérico de SDOs.	**5 de marzo**
Entrega del capítulo Implementación en Python de Métodos de Runge-Kutta encajados para la resolución de SDOs (+ correcciones del entregable anterior en paralelo).	**10 de abril**
Entrega del capítulo Análisis Numérico de Ecuaciones en Derivadas Parciales Elípticas (+ correcciones de entregables anteriores en paralelo).	**15 de mayo**
Entrega del capítulo Resolución de problemas de EDPs con Python (+ correcciones de entregables anteriores en paralelo).	**1 de junio**
Versión final de los 4 entregables anteriores.	**6 de junio**
Entrega del resumen y la Introducción.	**10 de junio**
Depósito del TFG en la plataforma correspondiente.	**13 de junio**

Cuadro 2.1: Distribución temporal de la entrega de los capítulos de la memoria.

Finalmente, entre el 13 y el 19 de junio, se reservan para las actividades previas a la defensa del proyecto, que incluyen la realización de una presentación con diapositivas para ilustrar la defensa, así como ensayos de la misma.

2.2.2. Plan de riesgos

En este apartado vamos a establecer los de planes de contingencia para hacer frente a los riesgos mencionados en 2.1.1, en caso de ser necesarios.

- Reorganización del cronograma. Dado que hay aproximadamente un mes de margen entre los entregables de mayor envergadura, se pueden readaptar plazos en caso de que haya retrasos considerables. Es más, en un caso muy extremo, el TFG se puede dejar para la Segunda Convocatoria Ordinaria.
- Para poder solventar las dificultades de implementación y análisis numérico, se recomienda buscar problemas similares que sirvan de inspiración y permitan que la alumna adquiera una mayor destreza resolviendo los problemas a los que tendrá que hacer frente. Del mismo modo la tutora se reunirá con la alumna para darle guía.
- Los problemas relacionados con la comprensión de la base matemática se resolverán mediante consultas a la tutora a lo largo del proceso.
- Por si el ordenador deja de ser funcional, la alumna ha previsto la posibilidad de realizar una petición de un portátil de préstamo de la US.

2.2.3. Plan de calidad

La calidad del proyecto estará definida por el cumplimiento de los siguientes indicadores:

- La nomenclatura y recursos formales matemáticos necesitados en cualquier parte del trabajo (aquí incluimos teoremas, proposiciones, demostraciones...) deben justificarse debidamente.
- La alumna deberá manejar los conceptos estudiados, de modo que pueda defender el contenido del trabajo.
- Los algoritmos deberán producir aproximaciones con escaso error en un tiempo computacional reducido.
- La presentación de código deberá ser limpia; con comentarios y una estructura cuidada y organizada, así como texto explicativo intercalado entre los distintos fragmentos.
- Deberán validarse los requerimientos iniciales indicados en 2.1.2.

2.2.4. Plan de comunicaciones

Para la comunicación entre la tutora y la alumna, se contemplan reuniones de seguimiento entre la cliente (tutora) y la jefa de proyecto (alumna) de forma periódica cada dos semanas. Asimismo, se utilizará el correo electrónico para realizar entregas, consultas y, en definitiva, cualquier labor de comunicación externa a las reuniones mencionadas.

3. Análisis Numérico de SDOs

En este capítulo se realiza una introducción teórica a la resolución numérica de un problema de Cauchy para un sistema diferencial ordinario, centrándonos en los métodos de un paso.

3.1. Análisis matemático de ecuaciones diferenciales ordinarias.

Recordamos en esta sección algunos de los principales resultados que se han visto en las asignaturas obligatorias *Ecuaciones diferenciales ordinarias* y *Ampliación de Ecuaciones Diferenciales ordinarias*.

Definición 3.1. *Consideramos un conjunto abierto $\Omega \subset \mathbb{R}^d$ y $T > 0$. Se dice que una función f es (globalmente) lipschitziana respecto de y en $[0,T] \times \Omega$, y se denota por $f \in \mathrm{Lip}_y([0,T] \times \Omega)$, si existe $L \geq 0$ tal que*

$$|f(t,y_1) - f(t,y_2)| \leq L|y_1 - y_2|, \qquad \forall\, (t,y_1),(t,y_2) \in [0,T] \times \Omega.$$

La función f se dice localmente lipschitziana respecto de y en $[0,T] \times \Omega$, y se denota por $f \in \mathrm{Lip}_{y,loc}([0,T] \times \Omega)$, si para todo $(t_0,y_0) \in [0,T] \times \Omega$ existe un entorno de este punto donde la función es globalmente lipschitziana respecto de y.

Observación 3.2. Si $f : [0,T] \times \Omega \to \mathbb{R}^d$ es tal que existe $D_y f \in C^0([0,T] \times \Omega; \mathcal{L}(\mathbb{R}^d))$, entonces $f \in \mathrm{Lip}_{y,loc}([0,T] \times \Omega)$. Esto nos aporta una condición suficiente para ser localmente Lipschitziana que suele verificarse por las funciones de este tipo más usuales.

En este capítulo vamos a estar interesados en resolver numéricamente el problema de Cauchy

$$\begin{cases} y'(t) = f(t,y(t)), & t \in [t_0, t_0+T], \\ y(t_0) = \eta. \end{cases} \tag{3.1}$$

donde $f \in C^0([t_0,t_0+T] \times \mathbb{R}^d)$, localmente Lipschitziana, es decir,

$$f \in Lip_y([t_0,t_0+T] \times \mathbb{R}^d)^d. \tag{3.2}$$

Vamos a definir el concepto de solución para este problema:

Definición 3.3. *Sea $I = I(t_0,\eta) \subseteq \mathbb{R}$ un intervalo tal que $t_0 \in \mathrm{int}(I)$, y sea $\varphi : I \to \mathbb{R}^d$. Diremos que φ es solución (local) del problema de Cauchy (3.1) en I si $\varphi \in C^1(I;\mathbb{R})$ y satisface*

1. *$(t,\varphi(t)) \in [0,T] \times \Omega$, para cualquier $t \in I$.*

2. $\varphi'(t) = f(t, \varphi(t))$, *para todo* $t \in I$.

3. $\varphi(t_0) = \eta$

Es usual escribir la solución del problema (3.1) no sólo como función de la variable t sino también como función de los datos iniciales. La solución correspondiente evaluada en (t, t_0, η) la notaremos

$$\varphi(t, t_0, \eta).$$

Definición 3.4. *Una solución φ del problema (3.1) se dice maximal si para otra solución $\psi \in C^1(I)$ con I un intervalo (abierto o cerrado) que tiene a t_0 como punto interior, tenemos*

$$I \subset I(t_0, y_0), \qquad \varphi(t) = \psi(t), \quad \forall\, t \in I.$$

En lugar de un problema de Cauchy para un sistema diferencial ordinario como es el caso de (3.1), podemos también considerar una ecuación de orden superior (más generalmente se podría considerar un sistema de orden superior) de la forma

$$\begin{cases} y^{d)} = g(t, y, y', y'', \ldots, y^{d-1)}) \ \text{ en un intervalo abierto } I \subset \mathbb{R}, \\ y(t_0) = \eta, \ y'(t_0) = \eta^1, \ \ldots \ , y^{d-1)}(t_0) = \eta^{d-1} \end{cases} \tag{3.3}$$

donde $g \in C^0(\Omega) \cap \mathrm{Lip}_{\mathrm{loc}}(y, \Omega; \mathbb{R})$. En realidad este caso se reduce al de sistemas haciendo el cambio de variables $y_1 = y$, $y_2 = y', \ldots y_N = y^{d-1)}$ que transforma el problema (3.3) en

$$\left\{ \begin{pmatrix} y_1' \\ y_2' \\ \vdots \\ y_N' \end{pmatrix} = \begin{pmatrix} y_2 \\ y_3 \\ \vdots \\ g(t, y_1, y_2, \ldots, y_N) \end{pmatrix}, \qquad \begin{pmatrix} y_1(0) \\ y_2(0) \\ \vdots \\ y_N(0) \end{pmatrix} = \begin{pmatrix} y_0 \\ y_0^1 \\ \vdots \\ y_0^{d-1} \end{pmatrix} \right.$$

Por tanto, tendría la misma formulación que (3.1) donde la función f es vectorial.

A continuación, presentamos algunos resultados estudiados en la asignatura *Ecuaciones diferenciales ordinarias* relativos a la existencia y unicidad de solución del problema (3.1):

Teorema 3.5. ***(Teorema de existencia y unicidad de solución maximal)*** *Sean $\Omega \subset \mathbb{R}^d$ abierto, $T > 0$, $f \in C^0([0,T] \times \Omega; \mathbb{R}^d) \cap f \in \mathrm{Lip}_{y,loc}([0,T] \times \Omega)$. Entonces, para todo $(t_0, \eta) \in [0,T] \times \Omega$ existe una única solución maximal $\varphi \in C^1(I(t_0, \eta_0))$ del problema de Cauchy (3.1). Además, el intervalo de existencia de la solución maximal, $I(t_0, \eta)$, es un intervalo abierto que contiene a t_0.*

Teorema 3.6. *Sea $\Omega = (a,b) \times \mathbb{R}^d$, con $-\infty \leq a < b \leq \infty$ y sea $f \in C^0(\Omega)^d$ tal que $f \in Lip(y; [c,d] \times \mathbb{R}^d)$, para todo $[c,d] \subset (a,b)$. Entonces, $\forall (t_0, y_0) \in \Omega$, la solución maximal del problema (3.1) está definida en todo el intervalo (a,b).*

Por norma general, no es posible calcular explícitamente la solución del problema (3.1), aunque se han visto en el grado algunos casos particulares interesantes en los que sí existe una forma analítica de calcular dichas soluciones. Un caso especialmente interesante (y simple) de sistemas es el de los sistemas lineales que son de la forma

$$\begin{cases} y' = A(t)y + b(t) \\ y(t_0) = y_0 \end{cases} \tag{3.4}$$

con $A \in C^0(I; \mathcal{L}(\mathbb{R}^d))$, $b \in C^0(I; \mathbb{R}^d)$, siendo I un intervalo abierto de $\mathbb{R}$. El sistema se dice homogéneo si b es nula.

Teorema 3.7. *Bajo las condiciones establecidas para el sistema (3.4), se tiene:*

a) Las soluciones maximales de (3.4) están definidas en todo el intervalo I.

b) El conjunto de soluciones del sistema homogéneo

$$y' = A(t)y \tag{3.5}$$

es un espacio vectorial de dimensión d.

c) El conjunto de soluciones del sistema no homogéneo

$$y' = A(t)y + b(t) \tag{3.6}$$

es un espacio afin de dimensión d tal que el espacio vectorial asociado coincide con el conjunto de soluciones del sistema homogéneo. Esto significa que si φ_p es una solución particular del sistema no homogéneo entonces cualquier otra solución φ es de la forma

$$\varphi(t) = \varphi_p(t) + \varphi_h(t)$$

con φ_h una solución del sistema homogéneo.

Si $\varphi_1, \cdots, \varphi_d$ es una base del espacio de soluciones del espacio de soluciones del sistema homogéneo, cualquier otra solución se podrá escribir como

$$\varphi(t) = c_1\varphi_1(t) + \cdots + c_d\varphi_d(t),$$

con $c_1, \ldots, c_d \in \mathbb{R}$ lo que se puede escribir en forma matricial como

$$\varphi(t) = F(t)c$$

siendo $F(t)$ la matriz que tiene por columnas las funciones $\varphi_1, \cdots, \varphi_N$ y c el vector columna de componentes $c_1, \ldots, c_N$. Esta función matricial F es lo que se conoce como una matriz fundamental del sistema.

Proposición 3.8. Una función $F \in C^1(I; \mathcal{L}(\mathbb{R}^d))$ es una matriz fundamental si y solo si verifica $F' = A(t)F$ en I y existe $t_0 \in I$ tal que $\det(F(t_0)) \neq 0$. Además, en este caso, se tiene

$$\det(F(t)) \neq 0, \text{ para todo } t \in I.$$

En el caso en que se conoce una matriz fundamental, o equivalentemente, se sabe resolver el sistema homogéneo, podemos calcular la solución del sistema no homogéneo mediante el método de variación de constantes que nos dice que las soluciones φ de (3.5) son de la forma

$$\varphi(t) = F(t)c(t)$$

donde c es ahora una función vectorial que verifica

$$F(t)c'(t) = b(t).$$

Sin embargo, nuestro objetivo va a ser resolver algunas situaciones realistas que corresponden matemáticamente a problemas del tipo (3.1), de las cuales no conocemos una expresión analítica de la solución. Por ello, nos centramos en estudiar métodos numéricos que nos permiten calcular una aproximación de la solución.

3.2. Análisis Numérico de métodos de un paso

Hay distintas técnicas que permiten obtener métodos numéricos con buenas propiedade para resolver el problema de Cauchy (3.1). En nuestro caso, estamos interesados en los llamados métodos de un paso. Estos métodos permiten aproximar la solución del problema en un conjunto de puntos, utilizando un algoritmo recurrente de un paso. Estos métodos además proporcionan una buena precisión y son relativamente simples de implementar.

Descripción del método:

Consideremos una partición $t_0 < t_1 < ... < t_n < t_{n+1} < ... < t_N$ del intervalo $[t_0, t_0 + T]$. Definiremos:

$$h_n \coloneqq t_{n+1} - t_n \quad \text{y} \quad h \coloneqq \max_{0 \leqq n < N} h_n$$

Los métodos de un paso pretenden encontrar una solución aproximada del problema (3.1) calculando en cada instante de la partición una aproximación de $y(t_n)$, que denotaremos por y_n. Todos estos métodos se escribirán de la forma:

$$\begin{cases} y_0 = \eta_h \quad \text{dado}, \\ y_{n+1} = y_n + h_n \Phi(t_n, y_n; h_n), \quad n \geq 0. \end{cases} \tag{3.7}$$

donde $\Phi \in \mathcal{C}^0([t_0, t_0 + T] \times \mathbb{R}^d \times [0, h^*])$ $(h > 0)$ y solo depende de la función f.

Utilizando estos métodos, pretendemos encontrar sucesiones convergentes a la única solución del problema (3.1), por lo que antes de describir algunos de estos métodos, vamos a ver nociones básicas sobre consistencia, estabilidad y convergencia, ya que nos van a permitir evaluar la eficacia de dichos métodos.

3.2.1. Consistencia, estabilidad y convergencia

Tal y como mencionamos previamente, nuestro objetivo es resolver numéricamente el problema (3.1). Para probar la efectividad de los distintos métodos, necesitamos probar la convergencia de las sucesiones generadas por los distintos algoritmos cuando el diámetro de la partición, h, tienda a 0.

En ocasiones, probar directamente esta propiedad es una labor difícil y laboriosa, es por ello que definimos otras propiedades que nos proporcionarán esta convergencia. Concretamente, estudiaremos las propiedades de consistencia y estabilidad que nos proporcionarán dicha convergencia.

Además, estas propiedades son importantes por sí mismas. Si el método numérico aproxima al continuo, es de esperar que la solución verifique de forma aproximada el algoritmo, esta es la idea de la consistencia. Por otra parte, la estabilidad hace mención a la influencia de los errores de medición y redondeo en el resultado final. Si esta influencia es muy grande, podemos llegar a un resultado sin correlación alguna con la solución correcta.

Definición 3.9. *Diremos que el método (3.7) es consistente con el sistema diferencial (3.1) si para la solución $y(.)$ del sistema $y' = f(t, y)$ se tiene:*

$$\lim_{h\to 0} \sum_{n=0}^{N-1} |y(t_{n+1}) - y(t_n) - h_n \Phi(t_n, y(t_n); h_n)| = 0$$

A $y(t_{n+1}) - y(t_n) - h_n\Phi(t_n, y(t_n); h_n)$ le llamaremos error de consistencia en el instante t_n.

Observación 3.10. Intuitivamente, representa el error cometido en el paso n-ésimo al reemplazar la solución del problema (3.1) por el término correspondiente de la sucesión generada por (3.7).

Definición 3.11. *Sea y_n la sucesión generada por (3.7) y sea z_n la sucesión generada por (3.7) donde hemos añadido una perturbación ε_n, es decir:*

$$\begin{cases} z_0 \quad \text{dado}, \\ z_{n+1} = z_n + h_n\Phi(t_n, z_n; h_n) + \varepsilon_n, \quad n \geq 0. \end{cases}$$

Diremos que el método (3.7) es estable si existe una constante $M > 0$, independiente de h, tal que:

$$\max_{0\leqq n<N} |z_n - y_n| \leq M[|y_0 - z_0| + \sum_{n<N} |\varepsilon_n|]. \tag{3.8}$$

Observación 3.12. Esta noción de estabilidad indica que una pequeña perturbación en los datos se traduce también una pequeña perturbación en la solución, lo que es necesario para el tratamiento numérico del problema debido a la existencia de los errores de redondeo que suelen cometerse.

Es conveniente recordar el concepto de error de discretización, ya que estamos generando una cantidad finita de términos de la sucesión dada por (3.7). Intuitivamente, el error de discretización numérico es un tipo de error que ocurre cuando se aproxima una solución continua a un problema matemático utilizando un método numérico que proporciona una representación discreta de la solución. Se produce porque la solución exacta del problema es continua, mientras que la representación discreta es finita y, por lo tanto, no puede capturar todos los detalles de la solución continua.

Este error puede ser reducido al disminuir el tamaño de los pasos o elementos utilizados en el método numérico. Sin embargo, esto puede aumentar el costo computacional del método y, en algunos casos, puede llevar a una inestabilidad numérica.

La definición matemática de este concepto es la siguiente:

Definición 3.13. *Entendemos por error de discretización (en el paso n-ésimo) a la diferencia entre $y(t_n)$ e y_n, , donde $y(.)$ es la solución del problema (3.1) e y_n es el término correspondiente de la sucesión que nos da el método (3.7).*

Definición 3.14. *Diremos que el método (3.7) es convergente si cuando el límite de η_h cuando h tiende a 0 es η, entonces $\max_{0\leqq n<N} |y(t_n) - y_n|$ tiende a cero.*

Teorema 3.15. ***(Teorema de Lax)*** *Si el métodos de un paso (3.7) es estable y consistente, entonces es convergente.*

Demostración. Es trivial a partir de lo visto hasta ahora. Basta considerar $\varepsilon_n = y(t_{n+1}) - y(t_n) - h_n\Phi(t_n, y_n; h_n)$. De la consistencia del método, se tiene:

$$\sum_{0\leqq n<N-1} \varepsilon_n \text{ tiende a cero.}$$

Ahora tomemos $z_n = y(t_n)$. Claramente, z_n y ε_n verifican (3.11), luego de la estabilidad del método deducimos

$$\max_{0\leqq n<N} |y(t_n) - y_n| \leq M[|\eta_h - \eta| + \sum_{n<N} |\varepsilon_n|]$$

teniendo así el resultado. ∎

Como dijimos anteriormente, la diferencia entre los distintos métodos de un paso es la función Φ que define el método. A continuación, estudiaremos condiciones suficientes sobre Φ que nos permitirán asegurar la consistencia y estabilidad del método y, por el Teorema (3.15), la convergencia del mismo.

Proposición 3.16. Una condición necesaria y suficiente para que el método (3.7) sea consistente es:

$$\Phi(t, y; 0) = f(t, y) \qquad \forall t \in [t_0, t_0 + T], \quad \forall y \in \mathbb{R}^d$$

La prueba puede consultarse en el capítulo 5 del Crouzeix-Mignot [1, pág 98] y se basa en el desarrollo de Taylor de las distintas funciones.

Proposición 3.17. Una condición suficiente para que el método (3.7) sea estable es que Φ sea globalmente Lipschitziana con respecto a la segunda variable, más concretamente, que exista una constante $\Lambda > 0$ y $h^* > 0$ tales que

$$|\Phi(t, y; h) - \Phi(t, z; h)| \leq \Lambda |y - z|, \quad \forall t \in [t_0, t_0 + T], \quad \forall y, z \in \mathbb{R}^d, \forall h \in [0, h^*] \tag{3.9}$$

Además, en estas condiciones, se tiene (3.8), con $M = e^{\Lambda T}$.

La prueba es inmediata a partir del lema de Gronwall discreto, que puede encontrarse por ejemplo en el Lema 4.1 de [1]. Podemos concluir que, aplicando el teorema de Lax, tenemos el siguiente resultado:

Teorema 3.18. *Supongamos que la función Φ cumple $\Phi(t, y; 0) = f(t, y)$ y la condición de Lipschitz (3.9). Entonces, el método (3.7) es convergente.*

3.2.2. Orden de métodos de un paso. Estudio del error de discretización

Definición 3.19. *Sean $y(t)$ la única solución de (3.1) y $p > 0$ y supongamos que $y \in C^{p+1}[t_0, t_0 + T]$. Entonces diremos que el método (3.7) es de orden p, si existe $K > 0$ independiente de h tal que*

$$\sum_{n=0}^{N-1} |y(t_{n+1}) - y(t_n) - h_n \Phi(t_n, y(t_n); h_n)| \leq K h^p.$$

Teorema 3.20. *Si el método de un paso es estable y de orden p y $f \in C^p([t_0, t_0 + T] \times \mathbb{R}^d)$, tenemos*

$$\forall n \leq N, \quad |y(t_n) - y_n| \leq M[|\eta_n - \eta| + K h^p].$$

Como hicimos anteriormente, vamos a buscar condiciones suficientes sobre Φ que nos permitan determinar el orden del método. Para ello, vamos a introducir la siguiente notación:

$$\begin{cases} f^{(0)}(t, y) = f(t, y) \\ f^{(1)}(t, y) = D_t f(t, y) + D_y f(t, y) f(t, y) \\ \vdots \\ f^{(k)}(t, y) = D_t f^{(k-1)}(t, y) + D_y f^{(k-1)}(t, y) f(t, y) \end{cases} \tag{3.10}$$

Teorema 3.21. *Sea y la solución del problema $y'(t) = f(t, y)$ e $I_0 \subset \mathbb{R}$ un intervalo no reducible a un punto. Supongamos $y \in C^1(I_0)$. Si $f \in C^p(I_0 \times \mathbb{R}^d)$, entonces $y \in C^{p+1}(I_0)$ y se verifica:*

$$y^{(p+1)}(t) = f^{(p)}(t, y(t)) \quad \forall t \in I_0,$$

donde $f^{(l)}$ *se define* $\forall l \in [0, p-1]$ *por recurrencia del siguiente modo:*

$$\begin{cases} f^{(0)}(t,y) = f(t,y) \\ f^{(l+1)}(t,y) = D_t f^{(l)}(t,y) + D_y f^{(l)}(t,y) \cdot f(t,y) \end{cases} \tag{3.11}$$

Demostración. Razonamos por inducción en p. Para $p = 0$, el resultado no es más que la definición de solución de $y' = f(t,y)$.

Supongamos $p \in \mathbb{Z}$, $p \geq 0$, tal que si $f \in C^p(I_0 \times \mathbb{R}^d)$, entonces $y \in C^{p+1}(I_0)$,

$$y^{p+1}(t) = f^{(p)}(t, y(t)), \quad \forall\, t \in I_0.$$

Tenemos que probar que si $f \in C^{p+1}(I_0 \times \mathbb{R}^d)$ entonces la afirmación anterior es cierta cambiando p por $p+1$.

Teniendo en cuenta que la definición de $f^{(p+1)}$ sólo hace intervenir sumas y productos de derivadas parciales de f de orden a lo más $p+1$, deducimos que $f^{(p)} \in C^1(I_0 \times \mathbb{R}^d)$. Como $y \in C^{p+1}(I) \subset C^1(I_0)$, entonces la regla de la cadena nos dice que la función

$$t \to y^{p+1}(t) = f^{(p)}(t, y(t)), \quad \forall\, t \in I_0,$$

está en $C^1(I_0)$ (luego $y \in C^{p+2)}(I_0)$) y se tiene

$$\begin{aligned} y^{p+2}(t) &= D_t f^{(p)}(t, y(t)) + D_y f^{(p)}(t, y(t)) \cdot y'(t) \\ &= D_t f^{(p)}(t, y(t)) + D_y f^{(p)}(t, y(t)) \cdot f(t, y(t)) = f^{(p+1)}(t, y(t)), \quad \forall\, t \in I_0. \end{aligned}$$

Teorema 3.22. *Supongamos que* $f \in C^p([t_0, t_0+T] \times \mathbb{R}^d)$ *y que* Φ, $\frac{\partial \Phi}{\partial h}$,..., $\frac{\partial^{p-1}\Phi}{\partial h^{p-1}}$ *existen y son continuas en* $[t_0, t_0+T] \times \mathbb{R}^d \times [0, h^*]$. *Entonces, una condición necesaria y suficiente para que el método sea de orden al menos* p *es:*
$\forall (t,y) \in [t_0, t_0+T] \times \mathbb{R}^d$, *se cumpla:*

$$\begin{cases} \Phi(t,y;0) = f(t,y) \\ \dfrac{\partial \Phi}{\partial h}(t,y;0) = \dfrac{1}{2} f^{(1)}(t,y), \\ \vdots \\ \dfrac{\partial^{p-1}\Phi}{\partial h^{p-1}}(t,y;0) = \dfrac{1}{p} f^{(p-1)}(t,y). \end{cases} \tag{3.12}$$

Demostración. Sea $\varepsilon_n = y(t_{n+1}) - y(t_n) - h_n \Phi(t_n, y(t_n); h_n)$, y

$$\Psi_k(t,y) = \frac{1}{(k+1)!} f^{(k)}(t,y) - \frac{1}{k!} \frac{\partial^k \Phi}{\partial h^k}(t,y;0). \tag{3.13}$$

De la expresión del polinomio de Taylor, tenemos:

$$\varepsilon_n = \sum_{k=0}^{p-1} h_n^{k+1} \Psi_k(t_n, y(t_n)) + O(h_n^{p+1}).$$

Tengamos en cuenta que las condiciones (3.12) equivalen a $\forall k < p$, $\Psi_k(t,y) \equiv 0$.

$\Rightarrow$ Supongamos que se verifica (3.12), entonces se tiene

$$|\varepsilon_n| \leq Ch_n^{p+1} \leq Ch^p h_n$$

de donde

$$\sum_{0 \leqq n < N} |\varepsilon| \leq Ch^p \sum_n h_n = CTh^p;$$

Luego el método es de orden p, quedando probada esta implicación.

$\Leftarrow$ Supongamos ahora, por Reducción al Absurdo, que (3.12) no es una condición necesaria y que no se verifica. Entonces, existirá $k \in \mathbb{Z}$, $k < p$ tal que $\Psi_k(t,y) \not\equiv 0$. Por simplificar la notación, podemos suponer que $h_n = h$ constante. Entonces

$$\varepsilon_n = h^{k+1}\Psi_k(t_n, y(t_n)) + O(h^{k+2}),$$

de donde

$$\sum_{0 \leqq n < N} |\varepsilon_n| = h^k \sum_n h|\Psi_k(t_n, y(t_n))| + TO(h^{k+1}).$$

Por tanto, si el método es de orden p,

$$0 = \lim_{h \to 0} \frac{1}{h^k} \sum_{0 \leqq n < N} |\varepsilon_n| = \int_{t_0}^{t_0+T} |\Psi_k(t, y(t))|\, dt,$$

lo que implica $\Psi_k(t, y(t)) = 0$ para todo $t \in [t_0, t_0 + T]$ y para toda solución $y(.)$ de $y'(t) = f(t, y(t))$.

Aplicando la Proposición 3.16, se tiene $\Psi_k(t,y) \equiv 0$, lo que es una contradicción. Luego se tiene el resultado. ∎

El Teorema 3.20 nos proporciona una estimación del error de discretización que puede mejorarse mediante el estudio de su comportamiento asintótico. Para ello, vamos a ver algunos resultados cuya teoría puede encontrarse en [1, págs 101-104]. Vamos a suponer a partir de ahora que la partición $t_0 < t_1 < ... < t_n < t_{n+1} < ... < t_N$ se ha tomado de tal manera que:

$$h_n = h(\theta(t_n) + O(h)) \tag{3.14}$$

donde $\theta \in Lip([t_0, t_0 + T])$ cumpliendo $\forall t \in [t_0, t_0 + T] \quad 0 < \theta(t) \leq 1$.

Teorema 3.23. *Supongamos que el método (3.7) es estable y de orden $p \geq 1$, $f \in C^{p+1}([t_0, t_0 + T] \times \mathbb{R}^d)$, $\Phi \in C^{p+1}([t_0, t_0 + T] \times \mathbb{R}^d \times [0, h^*])$ $(h^* > 0)$ y que $\eta - \eta_h = O(h^p)$. Entonces, el error de discretización cumple:*

$$y(t_n) - y_n = h^p e_p(t_n) + (\eta - \eta_h)e_0(t_n) + O(h^{p+1}),$$

donde e_0 y e_p son las soluciones respectivas de los siguientes SDOs:

$$\begin{cases} e_0'(t) = D_y f(t, y(t))e_0(t), \\ e_0(t_0) = 1, \end{cases} \qquad \begin{cases} e_p'(t) = D_y f(t, y(t))e_p(t) + \Psi_p(t, y(t))\theta(t)^p, \\ e_p(t_0) = 0, \end{cases} \tag{3.15}$$

donde Ψ_p es la función definida por (3.13) con $k = p$.

La prueba de este resultado puede encontrarse, por ejemplo, en el capítulo 5 de [1].

Teorema 3.24. *Bajo las condiciones del Teorema 3.23, supongamos también que las funciones f y Ψ son $(p+r+1)$ veces derivables con derivada continua, que el paso $h_n = h$ es constante y que $\eta_h = \eta$. Entonces existen funciones e_p,e_{p+1},...,e_{p+r} tales que*

$$y(t_n) - y_n = h^p e_p(t_n) + h^{p+1} e_{p+1}(t_n) + ... + h^{p+r} e_{p+r}(t_n) + O(h^{p+r+1}). \tag{3.16}$$

Demostración. Consideremos

$$\begin{aligned} &y_{p+r}(t) = yp + r(t,h) = y(t) - h^p e_p(t) - ... - h^{p+r} e_{p+r}(t), \\ &y_{p-1}(t) = y(t), \\ &\varepsilon_{p+r}(t,h) = y_{p+r}(t+h,h) - y_{p+r}(t,h) - h\Phi(t, y_{p+r}(t,h); h), \end{aligned}$$

y definamos recursivamente las funciones e_{p+r} con $r \geq 0$ como:

$$\begin{cases} e'_{p+r}(t) = D_y f(t, y(t)) e_{p+r}(t) + \frac{1}{(p+r+1)!} \frac{\partial^{p+r+1}}{\partial h^{p+r+1}} \varepsilon_{p+r-1}(t,0), \\ e_{p+r}(0) = 0. \end{cases}$$

Como el método es de orden p, tenemos: $\varepsilon_{p-1}(t,h) = y(t+h) - y(t) - h\Phi(t, y(t); h) = O(h^{p+1})$. Supongamos que $\varepsilon_{p+r-1} = O(h^{p+r+1})$, entonces:

$$\varepsilon_{p+r}(t,h) = \varepsilon_{p+r-1}(t,h) - h^{p+r}(e_{p+r}(t+h) - e_{p+r}(t)) + h(\Phi(t, y_{p+r-1}(t); h) - \Phi(t, y_{p+r}(t); h))$$

De donde, por Taylor, deducimos:

$$\begin{aligned} \varepsilon_{p+r}(t,h) \quad = \quad & \frac{h^{p+r+1}}{(p+r+1)!} \frac{\partial^{p+r+1}}{\partial h^{p+r+1}} \varepsilon_{p+r-1}(t,0) \\ & - h^{p+r+1} e'_{p+r}(t) + h D_y \Phi(t, y_{p+r}(t); h) h^{p+r} e_{p+r}(t) + O(h^{p+r+2}), \end{aligned}$$

Luego $\varepsilon_{p+r}(t,h) = O(h^{p+r+2})$. Utilizando la ecuación diferencial verificada por e_{p+r} y la relación

$$D_y \Phi()$$

Sean ahora $y_{n,p+r} = y_{p+r}(t_n, h)$ y $\varepsilon_{n,p+r} = \varepsilon_{p+r}(t_n, h)$. Tenemos entonces:

$$y_{n+1,p+r} = y_{n,p+r} + h\Phi(t_n, y_{n,p+r}; h) + \varepsilon_{n,p+r}$$

$$\sum_{0 \leq n < N} |\varepsilon_{n,p+r}| \leq CTh^{p+r+1}.$$

De la estabilidad del método obtenemos:

$$|y_n - y_{n,p+r}| \leq CMTh^{p+r+1}.$$

Definición 3.25. *Se dice que un métodos de un paso es reversible cuando $y_{n+1} = y_n + h_n \Phi(t_n, y_n; h_n)$ equivale a $y_n = y_{n+1} - h_n \Phi(t_{n+1}, y_{n+1}; -h_n)$.*

La propiedad de reversibilidad es importante en los métodos de un solo paso para resolver ecuaciones diferenciales porque permite que el mismo método se pueda utilizar para avanzar o retroceder en el tiempo. Otra ventaja de los métodos reversibles es que suelen tener errores de truncamiento locales simétricos, lo que puede mejorar la precisión de la aproximación.

3.3. Construcción de métodos de un paso

La idea más simple para construir un método de orden p es construir Φ basándonos en desarrollos de Taylor, es decir,

$$\Phi(t, y; h) = f(t, y) + \frac{h}{2} f^{(1)}(t, y) + ... + \frac{h^{p-1}}{p!} f^{(p-1)}(t, y), \tag{3.17}$$

ya que es trivial que las hipótesis (3.12) se verifican. Si las funciones $f^{(k)}(t, y)$, $k = 0, 1, ..., p-1$ verifican la condición de Lipschitz

$$\forall t \in [t_0, t_0 + T], \quad \forall y, z \in \mathbb{R}, \quad |f^{(k)}(t, y) - f^{(k)}(t, z)| \leq L_k |y - z|,$$

entonces, la función Φ la verificará también

$$\forall t \in [t_0, t_0 + T], \quad \forall y, z \in \mathbb{R}, \quad |\Phi(t, y; h) - \Phi(t, z; h)| \leq \Lambda |y - z|,$$

con

$$\Lambda = L_0 + \frac{h}{2} L_1 + ... + \frac{h^{p-1}}{(p-1)!} L_{p-1}. \tag{3.18}$$

Aplicando la Proposición 3.17, se tiene la estabilidad del método.

A pesar de que los métodos obtenidos al realizar desarrollos de Taylor verifican las hipótesis establecidas para los métodos de un paso, no son efectivos en la práctica. Por tanto, evitaremos estos métodos excepto en casos muy sencillos. A continuación, vamos a describir algunos métodos de interés, así como indicaremos sus propiedades y bajo qué condiciones se verifican estas.

3.3.1. Métodos de Runge-Kutta

Vamos a comenzar recordando el concepto de fórmula de cuadratura numérica que se vio en la asignatura *Cálculo Numérico I*:

Definición 3.26. *Sean $f : \mathbb{R}^d \to \mathbb{R}$, $a, b \in \mathbb{R}$. Una fórmula de cuadratura numérica es un método numérico para aproximar el valor de $\int_a^b f(x)dx$ mediante la evaluación de la función $f(x)$ en un conjunto finito de puntos $x_1, ..., x_n \in \mathbb{R}^d$ y la multiplicación de estos valores por unos pesos $w_1, ..., w_n \in \mathbb{R}$, y luego sumando los resultados. Es decir:*

$$\int_a^b f(x)\, dx \approxeq \sum_{i_1}^{n} w_i f(x_i)$$

Sean q números reales $c_1, c_2, ..., c_q$ conocidos, consideramos las siguientes fórmulas de cuadratura numérica

$$\int_0^{c_i} \Psi(t)\, dt \simeq \sum_{j=1}^{q} a_{ij} \Psi(c_j), \quad i = 1, 2, ..., q \tag{3.19}$$

y

$$\int_0^1 \Psi(t)\,dt \simeq \sum_{j=1}^{q} b_j \Psi(c_j). \tag{3.20}$$

Consideremos ahora los instantes de tiempo intermedios $t_{n,i} = t_n + c_i h_n$. Si la función y es solución de $y'(t) = f(t, y)$, tendremos al aplicar (3.19) y (3.20):

$$y(t_{n,i}) = y(t_n) + \int_{t_n}^{t_{n,i}} f(t, y(t))\,dt \simeq y(t_n) + h_n \sum_{j=1}^{q} a_{ij} f(t_{n,j}, y(t_{n,j})),$$

$$y(t_{n+1}) \simeq y(t_n) + h_n \sum_{j=1}^{q} b_j f(t_{n,j}, y(t_{n,j})), \quad y(t_{n,j}) \simeq y_{n,j} \quad \text{y} \quad y(t_{n+1}) \simeq y_{n+1}$$

Los métodos de Runge-Kutta simplemente consisten en calcular las aproximaciones mediante el siguiente esquema:

$$y_{n,i} = y_n + h_n \sum_{j=1}^{q} a_{ij} f(t_{n,j}, y_{n,j}), \quad i = 1, ..., q \tag{3.21}$$

$$y_{n+1} = y_n + h_n \sum_{j=1}^{q} b_j f(t_{n,j}, y_{n,j}), \quad i = 1, ..., q \tag{3.22}$$

con

$$t_{n,i} = t_n + c_i h_n, \quad i = 1, ..., q. \tag{3.23}$$

Supongamos conocida la aproximación y_n de $y(t_n)$. Los sistemas (3.21) forman a su vez un sistema de q ecuaciones con q incógnitas $y_{n,i}$. Los métodos de Runge-Kutta pueden ser considerados como métodos de un paso de la forma (3.7), donde la función $\Phi(.,.;.)$ está definida por:

$$\begin{cases} y_i = y + h \sum_{j=1}^{q} a_{ij} f(t + c_j h, y_j), & i = 1, ...q \\ \Phi(t, y; h) = \sum_{j=1}^{q} b_j f(t + c_j h, y_j). \end{cases} \tag{3.24}$$

Un método de Runge-Kutta está definido por el valor de q, de los coeficientes a_{ij}, b_j y c_j. De forma estándar estos métodos suelen representarse con la siguiente tabla:

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1q} \\ c_2 & a_{21} & a_{22} & \dots & a_{2q} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_q & a_{q1} & a_{q2} & \dots & a_{qq} \\ \hline & b_1 & b_2 & \dots & b_q \end{array}$$

A esta representación se la conoce como tablero de Butcher del método.

Definición 3.27. *Si la matriz formada por los coeficientes a_{ij} es estrictamente triangular inferior, se dice que el método de Runge-Kutta es explícito. Si dicha matriz es triangular inferior, diremos que el método es semi-implícito y, en el resto de casos, se dice que el método es implícito.*

En el caso explícito, los métodos de Runge-Kutta están siempre bien definidos y son fácilmente programables. Para el caso semi-implícito e implícito el cálculo de las aproximaciones requiere de la resolución de ecuaciones no lineales que no siempre tienen solución única. Normalmente, estos métodos están bien definidos bajo ciertas condiciones sobre h.

A continuación, indicaremos las propiedades de los métodos de Runge-Kutta y bajo qué condiciones se verifican. Para ello, vamos a comenzar introduciendo la siguiente notación:

$$A = \begin{pmatrix} a_{11} & a_{12} & ... & a_{1q} \\ a_{21} & a_{22} & ... & a_{2q} \\ \vdots & \vdots & & \vdots \\ a_{q1} & a_{q2} & ... & a_{qq} \end{pmatrix}, \quad C = \begin{pmatrix} c_1 & 0 & ... & 0 \\ 0 & c_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & ... & 0 & c_q \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_q \end{pmatrix}, \quad e = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix},$$

$$b^T = (b_1, b_2, ..., b_q). \tag{3.25}$$

Denotemos ahora por $|A|$ a la matriz resultante de tomar valor absoluto en cada uno de los coeficientes de la matriz A y consideremos $\rho(|A|)$ su radio espectral.

Proposición 3.28. Si f es globalmente Lipschitziana, entonces:

- Si $h_n L\rho(|A|) < 1 \ \forall n$, el método de Runge-Kutta admite una única solución.
- Si $h^* L\rho(|A|) < 1$, el método es estable para todo h tal que $0 < h \leq h^*$.
- Si $f(t, y) \in C^k([t_0, t_0 + T] \times \mathbb{R}^d)$, se tiene también que $\Phi(t, y; h) \in C^k([t_0, t_0 + T] \times \mathbb{R}^d \times [0, h^*])$.

Observación 3.29. Si el método de Runge-Kutta es explícito, $\rho(|A|) = 0$, por lo que el método siempre tendrá solución y, además, será estable.

En lo sucesivo, supondremos, por simplificar, que se verifica la siguiente condición:

$$Ae = Ce, \tag{3.26}$$

que simplemente viene a decir que las fórmulas de cuadratura (3.19) son exactas para polinomios de grado cero. Estudiemos ahora el problema del orden de consistencia para un método Runge-Kutta. Definimos las condiciones $A'(l)$ para $l \leq 4$:

l	$A'(l)$
1	$b^T e = 1$
2	$b^T Ce = b^T Ae = \frac{1}{2}$
3	$b^T C^2 e = b^T CAe = b^T (Ae)^2 = \frac{1}{3}$ $b^T ACe = b^T A^2 e = \frac{1}{6}$
4	$b^T C^3 e = b^T C^2 Ae = b^T C(Ae)^2 = b^T (Ae)^3 \frac{1}{4}$ $b^T CACe = b^T CA^2 e = b^T((Ae)\dot{(}ACe)) = b^T((Ae)\dot{(}A^2 e)) = \frac{1}{8}$ $b^T AC^2 e = b^T ACAe = b^T A(Ae)^2 = \frac{1}{12}$ $b^T A^2 Ce = b^T A^3 e = \frac{1}{24}$

(3.27)

Diremos que la condición $A(p)$ se verifica si se verifican las condiciones $A'(l)$, $\forall l \in 1 \leq l \leq p$.

Teorema 3.30. *Una condición necesaria y suficiente para que un método de Runge-Kutta sea de orden al menos p para toda función f suficientemente regular, es que esta verifique la condición $A(p)$.*

Observación 3.31. Realmente, se pueden definir condiciones de este tipo para $l > 4$, pero debido a que el número de condiciones crece mucho cuanto más grande sea el valor de l, y a que no se suelen utilizar métodos con orden de consistencia superior a 4 (estos ya son lo suficientemente buenos), no hemos incluido más. Puede encontrarse la formulación de estas condiciones en general, así como la prueba del resultado anterior en [1, págs 109-111]

Por último, vamos a ver algunos ejemplos de métodos de Runge-Kutta para terminar esta sección. Vamos a comenzar por ver los Métodos de Euler como casos particulares de Métodos de Runge-Kutta, ya que son los más sencillos de construir.

Ejemplo 3.32. Método de Euler explícito:

$$\begin{cases} y_{n+1} = y_n + h_n f(t_n, y_n), & n = 0, 1, ..., N-1, \\ y_0 = \alpha. \end{cases} \tag{3.28}$$

El método (3.28) no es más que el método de Runge-Kutta definido por $(q = 1)$ $\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}$, ya que nos da:

$$\begin{cases} y_{n,1} = y_n + 0, \\ y_{n+1} = y_n + h_n f(t_n, y_{n,1}) \end{cases}$$

Ejemplo 3.33. Método de Euler implícito o retrógrado

$$\begin{cases} y_{n+1} = y_n + h_n f(t_{n+1}, y_{n+1}), & n = 0, 1, ..., N-1, \\ y_0 = \alpha. \end{cases} \tag{3.29}$$

El método (3.29) corresponde al esquema de Runge-Kutta definido por $(q = 1)\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}$, que es de orden 1.

Ejemplo 3.34. Los métodos de Runge-Kutta definidos por $\begin{array}{c|cc} 0 & 0 & 0 \\ \alpha & \alpha & 0 \\ \hline & 1-\frac{1}{2\alpha} & \frac{1}{2\alpha} \end{array}$, con α un parámetro no nulo, se escriben como:

$$y_{n+1} = y_n + h_n(1 - \frac{1}{2\alpha}) f(t_n, y_n) + h_n \frac{1}{2\alpha} f(t_n + \alpha h_n, y_n + \alpha h_n f(t_n, y_n)).$$

Para $\alpha = \frac{1}{2}$, este método se conoce como el método de Euler modificado; para $\alpha = 1$, se conoce como método de Heun. Estos métodos pueden probarse que son de orden dos utilizando las condiciones $A'(l)$.

Ejemplo 3.35. El método de Crank-Nicolson, que es un método implícito de orden 2, se representa como:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

Ejemplo 3.36. El método de Runge-Kutta clásico, que es de orden 4, viene dado por el siguiente tablero:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

3.3.2. Control del paso.

Es importante controlar el tamaño de h_n $\forall n$ en todo instante de la partición, ya que pueden producirse grandes errores. Por ejemplo, consideremos el siguiente problema:

$$\begin{cases} y' = y, \text{ en } [0,T], \\ y(0) = 1. \end{cases}$$

Claramente, su solución es $y = e^t$ en $[0,T]$. A continuación, vamos a tratar de resolverlo mediante el método (3.28) con un paso constante h. El método de Euler consiste en tomar

$$y_0 = 1, \quad y_{n+1} = y_n + hy_n = (1+h)\, y_n, \ 0 \leq n \leq m-1$$

Se trata de una sucesión geométrica, por lo que

$$y_n = (1+h)^n .$$

Tomemos $T = 6$ y $h = 1$. Vamos a representar la aproximación que nos da el método junto con la solución analítica.

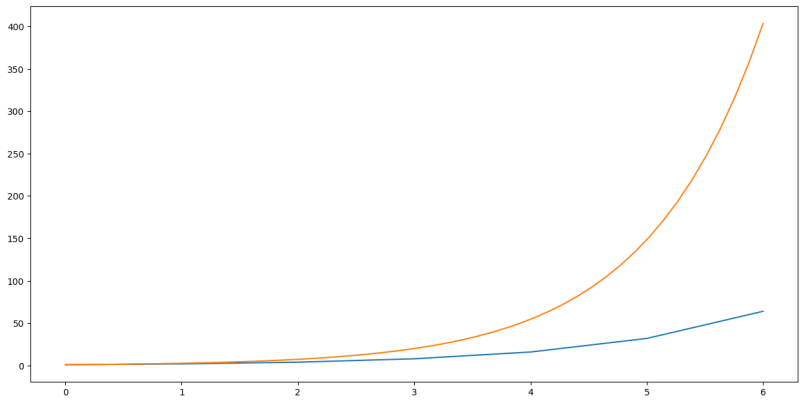

Figura 3.1: Comparación entre la exponencial y la aproximación dada por el método de Euler

Observamos que en intervalos en los que la pendiente de la exponencial no es muy grande, se obtiene suficiente precisión. Sin embargo, conforme dicha pendiente aumenta, se producen grandes errores, por lo que es necesario tomar un paso más pequeño conforme el tiempo avanza. Si bien es cierto que podría tomarse el paso pequeño y constante en todo el intervalo $[0, 6]$, se estarían realizando cálculos innecesarios desde el punto de vista computacional, pues ya hemos comprobado que con un paso mayor se obtienen buenos resultados para ciertos intervalos. Vemos, por tanto, la necesidad de controlar el paso.

El problema que nos planteamos es, para un N dado, cómo escoger los puntos t_n de forma que el error final $|y(t_N) - y_N|$ sea menor que una tolerancia prefijada. Las consideraciones previas nos animan a reemplazar este problema por: fijado $\frac{1}{h}\displaystyle\int_{t_0}^{t_0+T} \frac{1}{\theta(s)}\, ds$, encontrar h y θ tal que $h|z_1(t_N)|$ sea mínimo.

Remarquemos que solamente interviene $\phi = h\theta$, lo que reduce el problema a

$$\begin{cases} \text{encontrar} \quad \phi > 0 \quad \text{minimizando} \quad \left| \displaystyle\int_{t_0}^{t_0+T} \frac{y''(s)\phi(s)}{z_0(s)}\, ds \right| \\ \text{con} \quad \displaystyle\int_{t_0}^{t_0+T} \frac{1}{\phi(s)} \quad \text{dado.} \end{cases} \tag{3.30}$$

Vemos que si y'' es continua y no se anula en $[t_0, t_0 + T]$ (y, por tanto, tenga signo constante) el problema (3.30) admite una solución y una única

$$\phi(t) = \lambda \sqrt{\frac{z_0(t)}{|y''(t)|}}, \quad \text{con} \quad \lambda = \frac{\displaystyle\int_{t_0}^{t_0+T} \sqrt{\frac{|y''(s)|}{z_0(s)}}\, ds}{M}. \tag{3.31}$$

Entonces se tiene

$$\varepsilon_n = \frac{h_n^2}{2} y''(t_n) + O(h_n^3) = \frac{(h\theta(t_n))^2}{2} y''(t_n) + O(h_n^3) = \pm\lambda^2 z_0(t_n) + O(h_n^3)$$

lo que nos muestra que $\frac{|\varepsilon_n|}{z_0(t_n)}$ se mantiene (asintóticamente) estable.

Consideremos:

- Un método de un paso de orden p dado por $\Phi(t, y; h)$ con error de consistencia

$$\varepsilon_n = y(t_{n+1}) - y(t_n) - h_n \Phi(t_n, y(t_n); h_n).$$

- Un método de un paso de orden al menos $p + 1$ dado por $\Phi^*(t, y; h)$ con error de consistencia

$$\varepsilon_n^* = y(t_{n+1}) - y(t_n) - h_n \Phi^*(t_n, y(t_n); h_n)$$

El control del paso necesita el cálculo de una aproximación $\tilde{\varepsilon}_n$ de ε_n. que vendrá definido por la función. Sabemos que $\varepsilon_n^* = O(h_n^{p+2})$. Comparemos los errores de

consistencia de ambos métodos suponiendo $\Phi(t,y;h)$ y $\Phi^*(t,y;h)$ suficientemente regulares:

$$\begin{aligned}\varepsilon_n - \varepsilon_n^* &= h_n[\Phi^*(t_n, y(t_n); h_n) - \Phi(t_n, y(t_n); h_n)] \\ &= h_n[\Phi^*(t_n, y_n; h_n) - \Phi(t_n, y_n; h_n)] \\ &\quad + h_n \int_{y_n}^{y(t_n)} [D_y\Phi^*(t_n, r; h_n) - D_y\Phi(t_n, r; h_n)]\, dr.\end{aligned}$$

Como $y(t_n) - y_n = O(h^p)$, deducimos

$$\begin{aligned}\varepsilon_n &= h_n[\Phi^*(t_n, y_n; h_n) - \Phi(t_n, y_n; h_n)] + \\ &\quad h_n[D_y\Phi^*(t_n, c_n; 0) - D_y\Phi(t_n, c_n; 0)](y(t_n) - y_n) + h_n^2 O(h_n^p).\end{aligned}$$

Siendo los dos métodos de orden al menos 1, sabemos que se verifica

$$D_y\Phi^*(t_n, y; 0) - D_y\Phi(t_n, y; 0) = D_y f(t,y) - D_y f(t,y) = 0,$$

lo que, junto con la Lipschitzianidad de Φ^* con respecto a h, prueba

$$h_n \int_{y_n}^{y(t_n)} [D_y\Phi^*(t_n, r; h_n) - D_y\Phi(t_n, r; h_n)]\, dr = h_n^2 O(h_n^p).$$

Por tanto, se tiene

$$\varepsilon_n = \tilde{\varepsilon}_n + h_n^2 O(h^p), \tag{3.32}$$

con $\tilde{\varepsilon}_n = h_n[\Phi^*(t_n, y_n; h_n) - \Phi(t_n, y_n; h_n)]$.

A continuación, vamos a estudiar algunas técnicas de control del paso para el método de Euler explícito, que es de orden $p = 1$. Teniendo en cuenta que en este caso $\Phi(t,y,h) = f(t,y)$ no depende de h y suponiendo $f \in C^2([t_0, t_0+T] \times \mathbb{R}^d)$, se tiene que la función Ψ_1 definida por (3.13), verifica:

$$\Psi_1(t, y(t)) = \frac{1}{2} y''(t).$$

Por tanto, el Teorema 3.23 afirma

$$y(t_n) - y_n = h e_1(t_n) + (\eta - \eta_h) e_0(t_n) + O(h^2),$$

donde e_0, e_1 son las soluciones de

$$\begin{cases} e_0'(t) = D_y f(t, y(t)) e_0(t), \\ e_0(t_0) = 1, \end{cases} \qquad \begin{cases} e_1'(t) = D_y f(t, y(t)) e_1(t) + \frac{1}{2} y''(t)\theta(t)^1, \\ e_1(t_0) = 0. \end{cases}$$

Tomando $\eta - \eta_h = h\alpha + O(h^2)$, podemos entonces, gracias a la linealidad de las ecuaciones, escribir

$$y(t_n) - y_n = h g(t_n) + O(h^2),$$

donde $g \in C^1([t_0, t_0+T])$ es la solución del siguiente problema diferencial:

$$\begin{cases} g'(t) = D_y f(t, y(t)) g(t) + \frac{1}{2}\theta(t) y''(t), \quad t \in [t_0, t_0+T] \\ g(t_0) = \alpha \quad \alpha \in \mathbb{R}^d. \end{cases}$$

Este resultado puede utilizarse para la elección del paso h_n óptimo en cada instante. Gracias a (3.14), el número de pasos necesarios para llegar al instante $t_0 + T$ es

$$\begin{aligned} N &= \frac{h_0}{h_0} + \frac{h_1}{h_1} + ... + \frac{h_{N-1}}{h_{N-1}} = \frac{1}{h}\left[\frac{h_0}{\theta(t_0)} + \frac{h_1}{\theta(t_1)} + ... + \frac{h_{N-1}}{\theta(t_{N-1})}\right] + O(1) \\ &= \frac{1}{h}\int_{t_0}^{t_0+T} \frac{1}{\theta(s)}\, ds + O(1) \end{aligned}$$

Situémonos en el caso donde $m = 1$ (EDO). Sea z_0 la solución del problema de Cauchy

$$\begin{cases} z_0'(t) = D_y f(t, y(t)) z_0(t), \\ z_0(t_0) = 1. \end{cases}$$

Observemos que $\forall t \in [t_0, t_0 + T]$, $z_0(t) > 0$ gracias a la unicidad del problema de Cauchy. Utilizando el método de variación de constantes, obtenemos

$$g(t) = \alpha z_0(t) + \frac{1}{2}\int_{t_0}^{t} \frac{z_0(t) y''(s)\theta(s)}{z_0(s)}\, ds = \alpha z_0(t) + z_1(t)$$

de donde

$$y(t_n) - y_n = \alpha h z_0(t_n) + h z_1(t_n) + O(h^2). \tag{3.33}$$

Vemos así que la elección del paso, es decir, la elección de θ, no influye en el error $\alpha h z_0(t)$ provocado por la aproximación sobre las condiciones iniciales, en cambio interviene en el error debido a $h z_1(t)$.

La manera más sencilla de utilizar el método de Euler consiste en tomar un paso constante h, de forma que el error final sea inferior a una cierta tolerancia. Sin embargo, esto evidentemente solo puede llevarse acabo si se conoce la solución del problema a priori.

Utilizando (3.32) con $p = 1$ (el orden del método de Euler), se suelen utilizar varios tipos de control del paso. Nosotros vamos a describir la técnica que vamos a utilizar para este trabajo únicamente. Para más información, se puede consultar el capítulo 4 del Crouzeix-Mignot ([1]). Nuestro objetivo es hacer $\frac{|\varepsilon_n|}{z_0(t_n)}$ constante. Para ello damos un parámetro μ cercano a cero e independiente de h, calculamos en cada paso $\tilde{\varepsilon}_n$ y una aproximación $\tilde{z_0}(t_n)$ de $z_0(t_n)$ y ajustamos el paso h_n de forma que

$$\frac{|\tilde{\varepsilon}_n|}{\tilde{z_0}(t_n)} = \mu;$$

entonces se tiene por los cálculos realizados para obtener (3.32) en el particular del método de Euler ($p = 1$, $\Phi(t_n, y_n; h_n) = f(t_n, y_n)$)

$$\frac{h_n^2}{2}|y''(t_n)| + h_n^2 O(h) = \mu \tilde{z_0}(t_n).$$

Esta técnica podría llevarnos a errores donde y'' se anule, ya que h_n (y por tanto h) podría volverse muy grande y, por consiguiente, los términos $h_n^2 O(h)$ ya no serán

despreciables. Para evitar esto, elegiremos h_n lo más grande posible de forma que $\frac{|\tilde{\varepsilon}_n|}{\tilde{z_0}(t_n)} \leq \mu \quad \text{y} \quad h_n \leq 10\sqrt{\mu \tilde{z_0}(t_n)}$. Entonces se tiene

$$h = O(\sqrt{\mu}) \quad \text{y} \quad h_n = \sqrt{\mu}(\Psi(t_n) + O(h)),$$

con $\Psi(t) = inf\left\{\sqrt{\frac{2\tilde{z_0}(t)}{|y''(t)|}}, 10\sqrt{\tilde{z_0}(t)}\right\}$.

Observación 3.37. Observemos que la hipótesis (3.14) se verifica para $\theta(t) = \frac{\Psi(t)}{\max\limits_{s\in[t_0,t_0+T]}(\Psi(s))}$. Además, gracias a (3.33), si $\eta - \eta_h = O(\mu)$ se tiene

$$y(t_n) - y_n = \frac{1}{2}\sqrt{\mu} z_0(t_n) \int_{t_0}^{t_n} \frac{y''(s)\Psi(s)}{z_0(s)}\, ds + O(\mu).$$

No hemos hecho más que una descripción parcial de este tipo de control; $\tilde{\varepsilon}_n$ depende de h_n y resolver el problema con $\tilde{z_0} = 1$, $|\tilde{\varepsilon}_n| \leq \mu$ puede no ser simple y tampoco tiene por qué ser muy precisa. Podemos proceder así:

Partimos de un valor h_n definido en el paso anterior y calculamos y_{n+1} (o $y_{n+1}^{(p)}$) y $\tilde{\varepsilon}_n$:

- Si $|\tilde{\varepsilon}_n| > 1,2\mu$, sustituyo h_n por

  $$h_n h\left(\frac{0,9}{|\tilde{\varepsilon}_n|}\right)^{\frac{1}{2}}$$

  y recalculamos y_{n+1} y $\tilde{\varepsilon}_n$.
- Si $0,9\mu \leq \varepsilon_n \leq 1,1\mu$, tomo $h_{n+1} = h_n$.
- Si $|\tilde{\varepsilon}_n| < 0,9\mu$ ó $1,1\mu < |\tilde{\varepsilon}_n| \leq 1,2\mu$ tomo

  $$h_{n+1} = \text{mín}\left\{10h, \frac{h_n h}{|\tilde{\varepsilon}_n|^{\frac{1}{2}}}\right\}.$$

3.3.3. Métodos de Runge-Kutta encajados

El objetivo de estos métodos, usualmente conocidos como métodos DIRK (Diagonal Implicit Runge-Kutta methods), es utilizar el método de orden p para calcular la solución aproximada y el método de orden p' para estimar el error de consistencia para el control del paso. De este modo, podemos calcular el control del paso sin hacer un estudio asintotico del error. Estos métodos son especialmente útiles en la resolución de problemas stiff.

Los problemas stiff en el contexto de ecuaciones diferenciales ordinarias, son problemas cuyas soluciones constan de tres partes, una parte estacionaria, una transitoria lenta y otra transitoria rápida. El problema que nos encontramos en este tipo de ecuaciones es que el paso se elige por razones de estabilidad y no por precisión. Es

por ello que estamos interesados en los métodos de Runge-Kutta encajados, que son capaces de adaptar automáticamente el tamaño del paso de integración y nos permiten obtener una solución precisa y estable.

La idea de los pares encajados es simple, se basa en construir métodos de R-K que compartan las mismas etapas intermedias y solo difieran en el cálculo de y_{n+1} final, es decir, la diferencia es el vector b que define el método. Con esto se construyen dos métodos: uno de orden p y otro de orden p' con $p' > p$. La resolución del problema aproximado se hace mediante el método de orden p y el método de orden p' se utiliza para aproximar el error de consistencia que permite calcular el paso de integración.

Estos métodos se representan mediante el siguiente tablero de Butcher:

$$\begin{array}{c|ccc:c} c_1 & a_{11} & \dots & a_{1q} & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ c_q & a_{q1} & \dots & a_{qq} & 0 \\ \hdashline 1 & b_1 & \dots & b_q & 0 \\ \hline & b'_1 & \dots & b'_q & b'_{q+1} \end{array}$$

El b' correspondiente simplemente se calcula imponiendo a las matrices A y C del método las condiciones $A'(l)$ necesarias para que, gracias al Teorema 3.30, podamos asegurar que el método tiene orden al menos p'.

Estos métodos, a su vez, se dividen en distintas categorías (no excluyentes entre sí), que indicaremos mediante el uso de las siguientes siglas:

- S (singular): los elementos de la diagonal de la matriz A son iguales.
- E (primera fase explícita): la primera fila del tablero de Butcher es nula.
- Q (quasi): siempre se acompaña de la sigla S para expresar que el método es quasi-singular, es decir, todos los elementos diagonales son iguales excepto el primero o el segundo si este último es nulo (es decir, tenga la sigla S).

Presentamos, a continuación, el tablero de Butcher de métodos de Runge-Kutta con cuatro etapas de las diversas categorías de métodos DIRK para una mejor

comprensión:

c_1	a_{11}	0	0	0
c_2	a_{21}	a_{22}	0	0
c_3	a_{31}	a_{32}	a_{33}	0
c_4	a_{41}	a_{42}	a_{43}	a_{44}
	b_1	b_2	b_3	b_4
	b'_1	b'_2	b'_3	b'_4

DIRK

c_1	γ	0	0	0
c_2	a_{21}	γ	0	0
c_3	a_{31}	a_{32}	γ	0
c_4	a_{41}	a_{42}	a_{43}	γ
	b_1	b_2	b_3	b_4
	b'_1	b'_2	b'_3	b'_4

SDIRK

c_1	a_{11}	0	0	0
c_2	a_{21}	γ	0	0
c_3	a_{31}	a_{32}	γ	0
c_4	a_{41}	a_{42}	a_{43}	γ
	b_1	b_2	b_3	b_4
	b'_1	b'_2	b'_3	b'_4

QSDIRK

0	0	0	0	0
c_2	a_{21}	a_{22}	0	0
c_3	a_{31}	a_{32}	a_{33}	0
c_4	a_{41}	a_{42}	a_{43}	a_{44}
	b_1	b_2	b_3	b_4
	b'_1	b'_2	b'_3	b'_4

EDIRK

0	0	0	0	0
c_2	a_{21}	γ	0	0
c_3	a_{31}	a_{32}	γ	0
c_4	a_{41}	a_{42}	a_{43}	γ
	b_1	b_2	b_3	b_4
	b'_1	b'_2	b'_3	b'_4

ESDIRK

0	0	0	0	0
c_2	a_{21}	a_{22}	0	0
c_3	a_{31}	a_{32}	γ	0
c_4	a_{41}	a_{42}	a_{43}	γ
	b_1	b_2	b_3	b_4
	b'_1	b'_2	b'_3	b'_4

QUESDIRK

Para finalizar esta sección, vamos a ver algunos ejemplos de métodos encajados, que serán utilizados en el siguiente capítulo.

Ejemplo 3.38. *Método* RK_{12}. El método de orden 1 es el método de Euler.

0	0	0
1	1	0
	$\frac{1}{2}$	$\frac{1}{2}$

Ejemplo 3.39. *Método* RK_{24}. Es el método de Runge-Kutta clásico que vimos en el Ejemplo 3.36.

0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0
$\frac{1}{2}$	0	$\frac{1}{2}$	0	0
1	0	0	1	0
	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$

Ejemplo 3.40. El método de Bogacki-Shampine, que es de orden 3, está construído encajando un método de orden 2 sobre el método de Ralston de orden 3. Tanto el método de Bogacki-Shampine como los métodos de Ralston, pueden encontrarse en la sección 16.2 de [3].

0	0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	0
$\frac{3}{4}$	0	$\frac{3}{4}$	0	0	0
1	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$	0	0
1	$\frac{7}{24}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{8}$	0
	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$	0	0

Ejemplo 3.41. El método de Fehlberg, también conocido como RK_{45}.

$$
\begin{array}{c|cccccc:c}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{3}{8} & \frac{3}{32} & \frac{9}{32} & 0 & 0 & 0 & 0 & 0 \\
\frac{12}{13} & \frac{1932}{2197} & \frac{-7200}{2197} & \frac{7296}{2197} & 0 & 0 & 0 & 0 \\
1 & \frac{439}{216} & -8 & \frac{3680}{513} & \frac{-845}{4104} & 0 & 0 & 0 \\
\frac{1}{2} & \frac{-8}{27} & 2 & \frac{-3544}{2565} & \frac{1859}{4104} & \frac{-11}{40} & 0 & 0 \\
\hline
1 & \frac{25}{216} & 0 & \frac{1408}{2565} & \frac{2197}{4104} & \frac{-1}{5} & 0 & 0 \\
\hline
 & \frac{16}{135} & 0 & \frac{6656}{12825} & \frac{28561}{56430} & \frac{-9}{50} & \frac{2}{55} & 0
\end{array}
$$

4. Implementación en Python de Métodos de Runge-Kutta encajados para la resolución de SDOs

4.1. Implementación del código propuesto.

Para el desarrollo del algoritmo de resolución de SDOs que se propone en este trabajo, se ha necesitado de un entorno de programación con Python con los módulos numpy y matplotlib. En esta ocasión se ha optado por la instalación de Python en el sistema operativo anfitrión para utilizarlo como intérprete en la herramienta gráfica de programación Visual Studio Code. Los paquetes mencionados se han incluído mediante los siguientes comandos en la terminal del sistema:

- py -m pip install numpy
- py -m pip install matplotlib
- py -m pip install scipy

Esta combinación de paquetes es fundamental para cualquier proyecto con carga matemática que queramos realizar con Python, como es el caso de los algoritmos de inteligencia artificial, cuya base es la estadística, trabajos de numérico...

Numpy

Numpy es el paquete fundamental para la computación científica con Python, contiene, entre otros elementos, una gran variedad de herramientas propias de otros entornos como matlab, lo que nos permite utilizar, en particular, matrices de forma simple y en este trabajo es fundamental, ya que pretendemos resolver sistemas. Para más información se puede consultar su documentación oficial: https://numpy.org/doc/stable/.

Matplotlib

Matplotlib es la librería más conocida para realizar representaciones gracias a su compatibilidad con los vectores propios de numpy. Si bien existen otros módulos como seaborn, no nos van permitir tanta libertad como matplotlib, ya que están más pensados para aplicarse en problemas de Big Data, en los que tenemos dataframes con objetos caracterizados por diversas categorías a modo de base de datos y estas librerías justamente automatizan las relaciones entre categorías. La documentación oficial de matplotlib puede ser consultada en https://matplotlib.org/stable/.

Scipy

Para hacer las funciones de test de los métodos numéricos implementados, nos va a ser de utilidad el módulo interpolate de la librería scipy, que contiene varios métodos matemáticos implementados y, en particular, interpolación lineal. Para profundizar sobre scipy, consúltese la documentación oficial, que puede encontrarse en https://docs.scipy.org/doc/scipy/reference/index.html#scipy-api.

Una vez instalados numpy, matplotlib y scipy y aclarada su función, encabezaremos nuestro archivo rk.py con las instrucciones:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from scipy import interpolate
```

Esto nos permite utilizar todos los recursos matemáticos ofrecidos por numpy y por las sublibrerías pyplot y animation de la librería gráfica matplotlib; también hemos abreviado los nombres con lo que nos referiremos a dichos paquetes a la hora de programar.

Para una mayor claridad, es necesario dar contexto en caso de que el lector no esté muy familiarizado con la sintaxis de Python; en dicho lenguaje, cuando utilizamos un método perteneciente a una librería importada, es necesario incluir 'nombre_del_paquete.' antes de escribir el nombre del método en cuestión.

Asimismo, al utilizar 'as' al importar hemos escogido nombres distintos para referenciar los módulos, en lugar de usar su nombre por defecto. Podríamos haber escogido nombres abreviados cualquiera, pero estos son los más estandarizados entre la comunidad, lo que favorece la universalidad del programa. En cualquier caso se recomienda elegir un nombre breve para simplificar la labor de implementación.

Bajo esta configuración del entorno establecida, podemos empezar a desarrollar el código, cuya estructura va a dividirse en tres partes principalmente:

1. Implementación del método de Runge-Kutta encajado.
2. Almacén de métodos: guardamos tableros de Butcher de algunos métodos de Runge-Kutta encajados concretos.
3. Tests: realizamos tests de error que nos permiten validar los métodos en cuestión. Asimismo, efectuamos visualizaciones gráficas que facilitan la interpretación del resultado obtenido por el método correspondiente.

A continuación, vamos a estudiar estas partes una por una:

4.1.1. Método de Runge-Kutta (método encajado).

Utilizando la teoría expuesta en el capítulo 3, se construye el siguiente algoritmo para métodos de Runge-Kutta encajados, que comenzaremos por presentar a modo de pseudocódigo:

Datos de entrada:

- Matriz A. Matriz triangular inferior de orden q.
- Vector c correspondiente al soporte de las fórmulas de integración.
- Vector b con los pesos de la fórmula de integración.
- Vector $\hat{b}$ con los pesos de la fórmula de integración de forma superior.
- Escalar h, que representa el paso máximo.

Nota: Para programar, es más simple entender que el vector b no es más que la última fila de A.

Pasos: Fijado tol, que representa un número que sea pequeño respecto a h^{p+1} (por ejemplo h^{p+2}) siendo p el orden del método encajado, hacemos:

1. Se toma $h_0 = h$.
2. Calcular
$$t_{n,i} = t_n + c_i h_n, \quad 1 \leq i \leq q,$$
$$y_{n,i} = y_n + h_n \sum_{j=1}^{i-1} a_{ij} k_{n,j}, \quad k_{n,i} = f(t_{n,i}, y_{n,i}), \quad 1 \leq i \leq q$$
3. Si $a_{ii} \neq 0$ tomamos $u = y_{n,i}$, $v = u$, $y_{n,i} = u + h_n a_{ii} k_{n,i}$. Mientras $|y_{n,i} - v| > tol$
$$k_{n,i} = f(t_{n,i}, y_{n,i}), \quad v = y_{n,i}, \quad y_{n,i} = u + h_n a_{ii} k_{n,i}.$$
4. Se toma
$$\Phi_n = y_{n,q} - y_n, \quad \hat{\Phi}_n = h_n \sum_{i=1}^{q} \hat{b}_i k_{n,i}, \quad \varepsilon_n = |\hat{\Phi}_n - \Phi_n|.$$
Si $\varepsilon_n > 1,2\mu$ (donde $\mu = h^{p_1+1}$ con p_1 el menor de los órdenes de los 2 métodos de R-K necesarios para dar lugar al método encajado en cuestión), sustituyo h_n por
$$h_n h \Big(\frac{0,9}{\varepsilon_n}\Big)^{\frac{1}{p+1}}$$
y recomienzo el paso 2.

Tomo
$$y_{n+1} = y_n + \hat{\Phi}_n.$$
Si $0,9\mu \leq \varepsilon_n \leq 1,1\mu$, tomo $h_{n+1} = h_n$.

Si $\varepsilon_n < 0,9\mu$ ó $1,1\mu < \varepsilon_n \leq 1,2\mu$ tomo

$$h_{n+1} = \text{mín}\left\{10h, \frac{h_n h}{\varepsilon_n^{\frac{1}{p+1}}}\right\}.$$

La implementación en Python de dicho algoritmo es la siguiente:

```
def RungeKutta(funcion,t_0,t_f,y_0,A,bgorro,c,h,p1,p2):
    #establecemos la tolerancia
    tol=h**(p2+2)
    #almacenamos la longitud del vector c
    q=len(c)
    #calculamos mu
    mu=h**(p1+1)
    #almacenamos este valor para evitar repetir calculos
       innecesarios, ya que intervienen constantes
    var=h*0.9**(1/(p1+1))
    #inicializamos h_n,t_n e y_n
    h_n=h
    t_n=t_0
    y_n=y_0
    #evitamos que cada posicion se almacene como array
    if type(y_n)==np.ndarray:
        y_n=[y_n[i] for i in range(0,len(y_n))]
        valores=np.empty((0, len(y_n)))
    else:
        valores=[]
    #inicializamos la lista de tiempos
    tiempos=[]
    while(t_n<t_f):
        #inicializamos ep_n asegurandonos entrar en el
           bucle la primera vez
        ep_n=1.2*mu+1
        #actualizamos los tiempos y valores
        tiempos.append(t_n)
        valores=np.append(valores,[y_n],axis=0)
        while(ep_n>1.2*mu):
            t_ni=t_n+c[0]*h_n
            y_ni=y_n
            k_ni=funcion(t_ni,y_ni)
            k_n=np.array([k_ni])
            for i in range(1,q):
                t_ni=t_n+c[i]*h_n
                y_ni=y_n+h_n*sum([k_n[j]*A[i,j] for j in
                   range(0,i)])
                k_ni=funcion(t_ni,y_ni)
```

```
                #paso implicito
                if A[i,i]!=0:
                    cte=h_n*A[i,i]
                    u=y_ni
                    v=u
                    y_ni=u+cte*k_ni
                    while np.linalg.norm(y_ni-v)>tol:
                        k_ni=funcion(t_ni,y_ni)
                        v=y_ni
                        y_ni=u+cte*k_ni
                    #actualizo k_ni para la nueva y_ni
                    k_ni=funcion(t_ni,y_ni)
                #actualizamos k_n
                k_n=np.append(k_n,[k_ni],axis=0)
            #calculamos phi_n y phigorro_n
            phi_n=y_ni-y_n
            phigorro_n=h_n*sum([k_n[j]*bgorro[j] for j in
               range(0,q)])
            #calculamos epsilon_n
            ep_n=np.linalg.norm(phigorro_n-phi_n)
            if(ep_n>1.2*mu):
                h_n=h_n*var/ep_n**(1/(p1+1))
        #actualizamos el valor de y_n
        y_n=y_n+phigorro_n
        #evitamos que cada posicion se almacene como array
        if type(y_n)==np.ndarray:
            y_n=[y_n[i] for i in range(0,len(y_n))]
        t_n=t_n+h_n
        if not (0.9*mu<=ep_n and ep_n<=1.1*mu):
            h_n=min(10*h,h_n*h/ep_n**(1/(p1+1)))
    return (tiempos,valores)
```

Las principales dificultades que han surgido a la hora de implementar el algoritmo han estado relacionadas con el carácter matricial de las variables y_n y k_n, ya que se han querido almacenar los y_n como filas de la matriz valores y los $k_{n,i}$ como filas de la matriz k_n.

Para añadir filas a una matriz, se utiliza el método np.append, que nos recuerda al método append para añadir elementos a listas disponible en Python base. Sin embargo, este último añade y actualiza la lista original, mientras que np.append requiere almacenar la lista con el elemento añadido. Por tanto, se cometió el error de no actualizar las variables correspondientes, impidiendo que se añadieran las filas correspondientes. Una vez resuelto haciendo uso de la documentación oficial, surgió un nuevo problema: las matrices se actualizaban almacenando todos los elementos en la misma fila, dando lugar a vectores y no matrices. Esto se solucionó añadiendo el parámetro axis=0, que especifica que se añade como fila.

Además, al querer inicializar tiempos y valores como una lista y un array vacíos, hemos tenido que distinguir entre el caso de una ecuación y un sistema. Para un sistema, debemos inicializar valores como una matriz sin filas y tantas columnas como elementos tenga y_n para evitar que se produzcan errores dimensionales. Esto claramente carece de sentido si es una ecuación, pues y_n será escalar y no contendrá elementos (no se le puede aplicar el método len).

En ese caso, hemos optado por usar listas, al igual que con los tiempos y simplemente con un if hemos preguntado por el tipo. Otra posibilidad era no inicializar como vacíos tiempos y valores, sino ya conteniendo el instante y valor inicial, pero no se consideró una buena práctica, ya que en ese caso se tendrían almacenados dichos valores por duplicado y habría que borrar el primer elemento de cada contenedor al final.

Asimismo, ya se estaba haciendo esta distinción según el tipo (luego la estábamos aprovechando), pues los y_n se estaban almacenando como vectores por sí mismos al hacer las operaciones requeridas por el algoritmo, en lugar de listas. Por culpa de esto, valores quedaba como un array unidimensional cuyos elementos eran arrays en vez de una matriz.

Desde el punto de vista del algoritmo y de los resultados numéricos producidos, esto no suponía ningún problema más allá de la incomodidad visual para leer los resultados. No obstante, esto imposibilitaba las posteriores representaciones gráficas realizadas en los tests, ya que los datos de entrada no tenían un formato válido.

4.1.2. Almacén de métodos.

En esta parte del código almacenamos los métodos $R_{pp'}$ que se utilizarán en los diversos tests realizados a los problemas resueltos. Para esto, cada método tendrá asociado unos órdenes p_1 y p_2, que se corresponderán con p y p', respectivamente, una matriz A (que, tal y como explicamos al describir el algoritmo, es la matriz A del tablero de Butcher añadiéndole como última fila el vector b) y los vectores c y $\hat{b}$. En particular, hemos almacenado los siguientes métodos: Euler, Euler Implícito, Bogacki-Shampine, Runge-Kutta clásico y Fehlberg. El código correspondiente a esta parte es el siguiente:

```
def almacen(nombre):
    if nombre=="Heun":
        p1=1
        p2=2
        A=np.array([[0,0],[1,0]])
        c=np.array([0,1])
        b_gorro=np.array([0.5,0.5])
    elif nombre=="Heun-EulerImp":
        p1=1
```

```
        p2=2
        A=np.array([[0,0],[0,1]])
        c=np.array([0,1])
        b_gorro=np.array([0.5,0.5])
    elif nombre=="clasico":
        p1=2
        p2=4
        A=np.array
           ([[0,0,0,0],[0.5,0,0,0],[0,0.5,0,0],[0,0,1,0]])
        c=np.array([0,0.5,0.5,1])
        b_gorro=np.array([1/6,1/3,1/3,1/6])
    elif nombre=="Bogacki-Shampine":
        p1=2
        p2=3
        A=np.array([[0,0,0,0,0],[0.5,0,0,0,0],
                    [0,0.75,0,0,0],
                    [2/9,1/3,4/9,0,0],
                    [7/24,0.25,1/3,1/8,0]])
        c=np.array([0,0.5,0.75,1,1])
        b_gorro=np.array([2/9,1/3,4/9,0,0])
    elif nombre=="Fehlberg5":
        p1=4
        p2=5
        A=np.array([[0,0,0,0,0,0,0],[0.25,0,0,0,0,0,0],
                    [3/32,9/32,0,0,0,0,0],
                    [1932/2197,-7200/2197,7296/2197,0,0,0,0],

                    [439/216,-8,3680/513,-845/4104,0,0,0],
                    [-8/27,2,-3544/2565,1859/4104,-11/40,0,0],

                    [25/216,0,1408/2565,2197/4104,-1/5,0,0]])

        c=np.array([0,0.25,3/8,12/13,1,0.5,1])
        b_gorro=np.array
           ([16/135,0,6656/12825,28561/56430,-9/50,2/55,0])
    return(p1,p2,A,c,b_gorro)
```

Su integración con el resto de código es muy sencilla: la función almacén nos devolverá una tupla con los atributos que caracterizan a cada método ya mencionados. De este modo, dado un problema concreto y un método para resolverlo, la función resuelve_problema extraerá los datos de la tupla que nos devuelve el método almacén y los utilizará para invocar al algoritmo de Runge-Kutta usando el problema como función f tal y como podemos observar:

```
def resuelve_problema(tupla,problema,t_0,t_f,y_0,h=0.001):
    p1,p2,A,c,b_gorro=tupla
    tiempos,valores=RungeKutta(problema,t_0,t_f,y_0,A,
```

```
        b_gorro,c,h,p1,p2)
    return (tiempos,valores)
```

4.1.3. Tests.

En cuanto a los tests, diferenciaremos entre dos tipos:

- Tests encargados de hacer representaciones gráficas a modo de simulaciones de las órbitas/trayectorias. Dichas representaciones se mostrarán junto con el problema al que vayan asociadas.
- Tests encargados de medir errores y hacer comparaciones entre los distintos métodos y valores de h escogidos, lo que nos permite validar el orden de cada método particular y, en definitiva, su precisión.

  Debemos recordar que, por norma general, no vamos a ser capaces de calcular la solución exacta del problema 3.1, por lo que vamos a implementar un método test para problemas en los que conozcamos la solución y otro para el caso contrario.

  En el primer caso, calcularemos los errores como la norma infinito de la diferencia entre la solución real y la que nos devuelve el método escogido. En el segundo caso, utilizaremos el método de Fehlberg, que es de orden 5, como estimación a priori para el cálculo del error. Dado que el vector de tiempos de Runge-Kutta no será el mismo para Fehlberg que para cada método a comparar, haremos interpolación lineal a trozos para tener una estimación para todo tiempo del intervalo. Evidentemente, esto produce errores, pero no nos impiden poder comparar los métodos.

Mostramos los tests de medición de error a continuación:

```
def test_resoluble(problema,solucion,t_0,t_f,y_0):
    metodos=[("Heun",2)
             ,("Heun-EulerImp",2)
             ,("clasico",4)]
    h_val=[0.4,0.2,0.1,0.05,0.025]
    #generamos 3 graficas simultaneamente
    fig, axs = plt.subplots(5)
    fig.suptitle('Comparacion␣del␣error␣para␣los␣distintos␣
       metodos')
    axs[0].set_title("h="+str(h_val[0]))
    axs[1].set_title("h="+str(h_val[1]))
    axs[2].set_title("h="+str(h_val[2]))
    axs[3].set_title("h="+str(h_val[3]))
    axs[4].set_title("h="+str(h_val[4]))
    for metodo in metodos:
```

```
        print("=========================================="
            )
        for i in range(0,len(h_val)):
            orden=metodo[1]
            t,v=resuelve_problema(almacen(metodo[0]),
                problema,t_0,t_f,y_0,h_val[i])
            if(i==0):
                err=np.max(np.abs(v[:,0]-solucion(t)))
                print("El metodo",metodo[0]+", con h=",
                    h_val[0],"nos da un error de ",err,".")
            else:
                err_anterior=err
                err=np.max(np.abs(v[:,0]-solucion(t)))
                orden=np.log(err_anterior/err)/np.log(h_val
                    [i-1]/h_val[i])
                print("El metodo",metodo[0]+", que es de
                    orden ",orden,", con h=",h_val[i],"nos
                    da un error de ",err,".")
            #representamos el error en la grafica
                correspondiente
            axs[i].plot(t,np.abs(v[:,0]-solucion(t)),label=
                metodo[0])
            axs[i].legend(loc="upper left")
    plt.show()

def test_irresoluble(problema,t_0,t_f,y_0,metodos,href,
    h_val):
    #resolvemos el problema por fehlberg como referencia
    t1,sol=resuelve_problema(almacen("Fehlberg5"),problema,
        t_0,t_f,y_0,href)
    #hacemos interpolacion lineal para tener sol para todo
        t
    interp = interpolate.interp1d(t1, sol[:,0], kind = "
        linear")
    #generamos 3 graficas simultaneamente
    fig, axs = plt.subplots(len(h_val))
    fig.suptitle('Comparacion del error para los distintos
        metodos')
    for i in range(0,len(h_val)):
        axs[i].set_title("h="+str(h_val[i]))
    for metodo in metodos:
        print("=========================================="
            )
        for i in range(0,len(h_val)):
            t2,v=resuelve_problema(almacen(metodo),problema
                ,t_0,t_f,y_0,h_val[i])
```

```
            if t2[-1]>t1[-1]:
                for j in range(0,len(t2)):
                    if t2[j]>t1[-1]:
                        t2=t2[:j]
                        v=v[:j,:]
                        break
            if(i==0):
                err=np.max(np.abs(v[:,0]-interp(t2)))
                print("El metodo",metodo+", con h=",h_val
                   [0],"nos da un error de ",err,".")
            else:
                err_anterior=err
                err=np.max(np.abs(v[:,0]-interp(t2)))
                orden=np.log(err_anterior/err)/np.log(h_val
                   [i-1]/h_val[i])
                print("El metodo",metodo+", que es de orden
                   ",orden,", con h=",h_val[i],"nos da un
                   error de ",err,".")
            #representamos el error en la grafica
               correspondiente
            axs[i].plot(t2,np.abs(v[:,0]-interp(t2)),label=
               metodo)
            axs[i].legend(loc="upper left")
    plt.show()
```

4.2. Tests numéricos de validación

En esta sección presentaremos y resolveremos algunos problemas de SDOs utilizando el método descrito en la sección anterior.

4.2.1. Problema del péndulo.

Péndulo linealizado

Suponemos que el ángulo es pequeño y, por infinitésimos equivalentes, $sen(y)$ se aproxima por y. Esto nos da la ecuación del movimiento armónico simple.

$$y'' + \frac{g}{l}y = 0,$$

donde $g = 9,8\frac{m}{s^2}$ es la aceleración de la gravedad, l la longitud de la varilla del péndulo e y el ángulo que forma el péndulo con la vertical. Esta ecuación se sabe resolver analíticamente siendo la solución:

$$y = c_1 cos(\frac{g}{l}t) + c_2 sen(\frac{g}{l}t),$$

donde c_1 y c_2 se calculan a partir de las condiciones iniciales. La solución del problema es muy regular, de hecho, es de clase infinito, por tanto, es una buena candidata para validar el orden de los métodos planteados. El orden teórico puede ser aproximado mediante la expresión $p \approx \log_{\frac{h_1}{h_2}} \left(\frac{\text{error}(h_1)}{\text{error}(h_2)} \right)$, siendo h_1,h_2 dos pasos de mallas distintos y error(h) denota el error en norma infinito que se comete para la malla de paso h. En nuestro caso, hemos utilizado los valores de h que presentamos en la Tabla 4.1.

Vamos a suponer que partimos con un ángulo 0, con una velocidad 1 y $l = g$. Nos queda el sistema:

$$\begin{cases} y_1' = y_2 \\ y_2' = -y_1 \\ y_1(0) = 0 \\ y_2(0) = 1, \end{cases}$$

cuya solución analítica es $y = \sin(t)$. El código de resolución de este problema se escribe como:

```
def pendulo_linealizado(t,y):
    return np.array([y[1],-y[0]])
```

Con la siguiente línea, aplicamos el método test_resoluble previamente expuesto para obtener una estimación del error resolviendo el problema en $[0, 10\pi]$.

```
#test_resoluble(pendulo_linealizado,np.sin,0,4*np.pi,np.
   array([0,1]))
```

En la tabla 4.1 recogemos los resultados obtenidos para los siguientes métodos: Euler encajado con Heun, Euler implícito encajado con Heun y Runge-Kutta clásico.

Metodo	h	Error	Orden : $\log_{\frac{h_{anterior}}{h_{actual}}}(\frac{err_{anterior}}{err_{actual}})$
Euler − Heun	0,4	0,6381723498175382	
Euler − Heun	0,2	0,1565105261943175	2,027686412518976
Euler − Heun	0,1	0,041485045039988004	1,9155964320078716
Euler − Heun	0,05	0,010440270384404223	1,9904322768593612
Euler − Heun	0,025	0,0026067228982508163	2,001849941901333
Euler Implicito − Heun	0,4	0,9594246344311848	
Euler Implicito − Heun	0,2	0,3255631362599048	1,5592321315299138
Euler Implicito − Heun	0,1	0,0815926455760968	1,996426326889886
Euler Implicito − Heun	0,05	0,020748302630158658	1,9754458016891037
Euler Implicito − Heun	0,025	0,005218839187829186	1,991192464932518
Clasico	0,4	0,05457096847042919	
Clasico	0,2	0,0041152484095317125	3,729082223365723
Clasico	0,1	0,0002698169107685222	3,930926841843608
Clasico	0,05	$1{,}7680798299245237e-05$	3,931725452089847
Clasico	0,025	$1{,}1191993951669907e-06$	3,9816444217220655

Cuadro 4.1: Estimación del orden de los métodos.

Representando las trayectorias del resultado obtenido por los distintos métodos, observamos que se obtiene la función seno, tal y como se espera. Por ejemplo, vamos a mostrar la representación de la trayectoria obtenida tras aplicar el método clásico. Para ello usaremos el siguiente código:

```
def trayectoria2D(tiempos,valores):
    y=valores[:,0]
    x=tiempos
    plt.plot(x,y)
    plt.scatter([0,np.pi/4,np.pi/2,3*np.pi/4,np.pi,5*np.pi
       /4,6*np.pi/4,7*np.pi/4,
                 2*np.pi,2*np.pi+np.pi/4,2*np.pi+np.pi/2,2*
                    np.pi+3*np.pi/4,3*np.pi,3*np.pi+np.pi
                    /4,3*np.pi+np.pi/2,3*np.pi+3*np.pi/4,4*
                    np.pi],
                 [0,np.sqrt(2)/2,1,np.sqrt(2)/2,0,-np.sqrt
                    (2)/2,-1,-np.sqrt(2)/2,0,np.sqrt(2)
                    /2,1,np.sqrt(2)/2,0,-np.sqrt(2)/2,-1,-
                    np.sqrt(2)/2,0],c="red")
    plt.show()

t,v=resuelve_problema(almacen("clasico"),
   pendulo_linealizado,0,4*np.pi,np.array([0,1]))
trayectoria2D(t,v)
```

Esta es la gráfica resultante marcando puntos propios de la función seno cada paso de $\pi/4$:

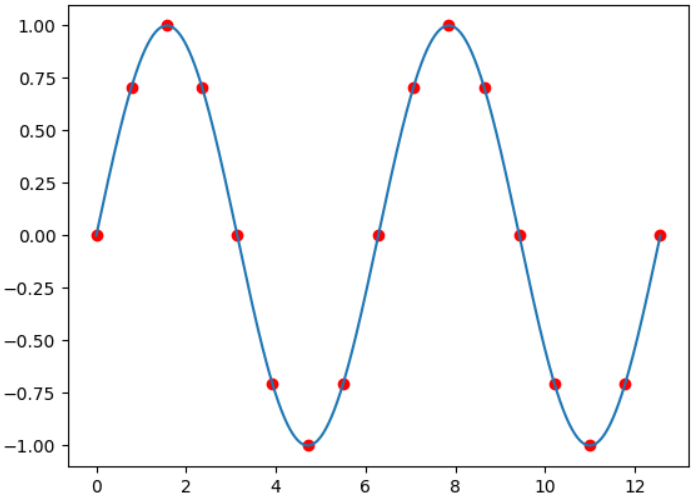

Figura 4.1: Trayectoria de la solución dada por el método clásico.

En la figura 4.2, representamos los errores cometidos para los distintos pasos de malla.

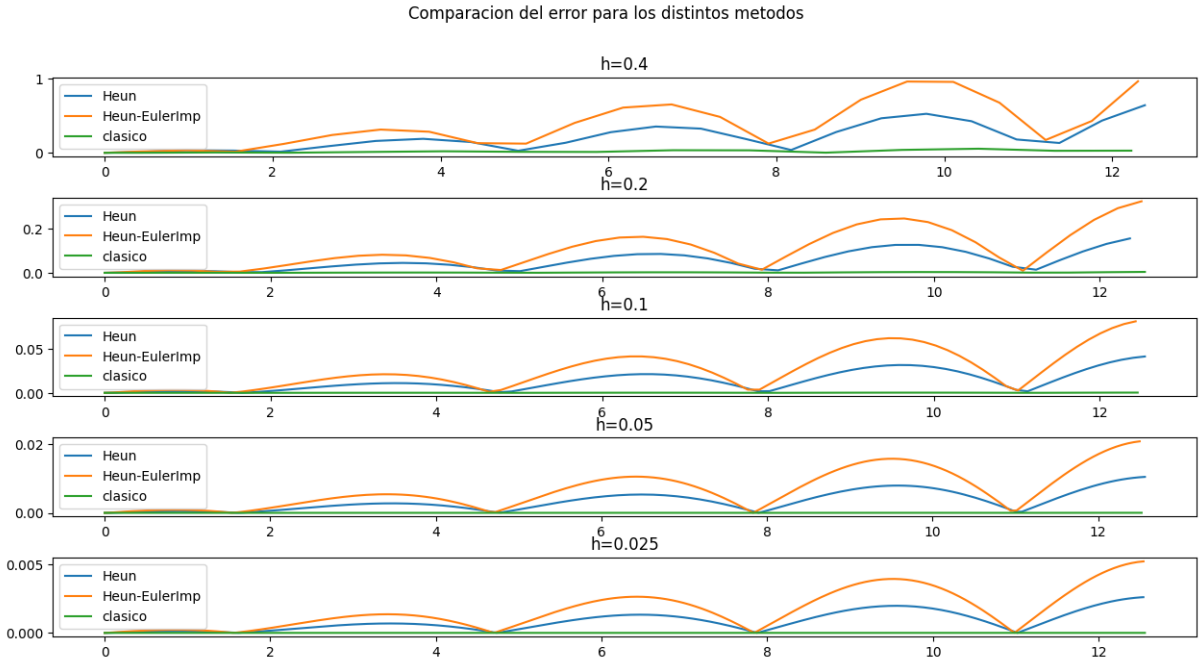

Figura 4.2: Errores cometidos en el problema del péndulo linealizado.

Una vez comprobado el orden teórico de los esquemas, pasamos a aproximar el problema del péndulo general donde no conocemos la solución analítica. En este

caso, vamos de nuevo a validar el orden de convergencia. Como no conocemos la expresión de la solución, utilizamos la aproximación que nos proporciona el método de Fehlberg como solución y repetimos el mismo proceso del caso anterior.

Péndulo generalizado

En general, el movimiento de un péndulo viene descrito por la siguiente ecuación diferencial:

$$y'' + \frac{g}{l} sen(y) = 0$$

donde, como en el caso linealizado, $g = 9,8\frac{m}{s^2}$ es la aceleración de la gravedad, l la longitud de la varilla del péndulo e y el ángulo que forma el péndulo con la vertical. Vamos a suponer que partimos con un ángulo 0, con una velocidad 1 y $l = g$. Pasándolo a sistema, nos queda:

$$\begin{cases} y_1' = y_2 \\ y_2' = -sen(y_1) \\ y_1(0) = 0 \\ y_2(0) = 1 \end{cases}$$

El código correspondiente a este problema es el siguiente:

```
def pendulo(t,y):
    return np.array([y[1],-np.sin(y[0])])
```

Mediante la siguiente línea, aplicamos el método test_irresoluble previamente expuesto para obtener una estimación del error ejecutando el método entre 0 y 5:

```
#test_irresoluble(pendulo,0,5,np.array([0,1]),["Heun","Heun
   -EulerImp","clasico"],0.001,[0.5,0.25,0.125,0.0675])
```

En la tabla 4.2 presentamos de nuevo los resultados obtenidos, donde comprobamos que, asintóticamente, la aproximación del orden de convergencia se acerca al orden teórico.

Metodo	h	Error	Orden : $\log_{\frac{h_{anterior}}{h_{actual}}}(\frac{err_{anterior}}{err_{actual}})$
Euler − Heun	0,5	0,20840921522745903	
Euler − Heun	0,25	0,05662724544633957	1,8798508119044015
Euler − Heun	0,125	0,013620458239020083	2,055721112251637
Euler − Heun	0,0675	0,003976966508264568	1,997859934079963
Euler Implicito − Heun	0,5	0,4547209126350142	
Euler Implicito − Heun	0,25	0,1241945801684976	1,872379139264782
Euler Implicito − Heun	0,125	0,03091675872029348	2,006141234244483
Euler Implicito − Heun	0,0675	0,009110248855632996	1,9830020256219298
Clasico	0,5	0,03168163669873092	
Clasico	0,25	0,0030902654981951327	3,3578441713301883
Clasico	0,125	0,0002256535791625991	3,7755492247957854
Clasico	0,0675	$2{,}01459016400396e-05$	3,9208933112673368

Cuadro 4.2: Estimación del orden de los métodos.

En la figura 4.3 vemos los errores obtenidos en este caso:

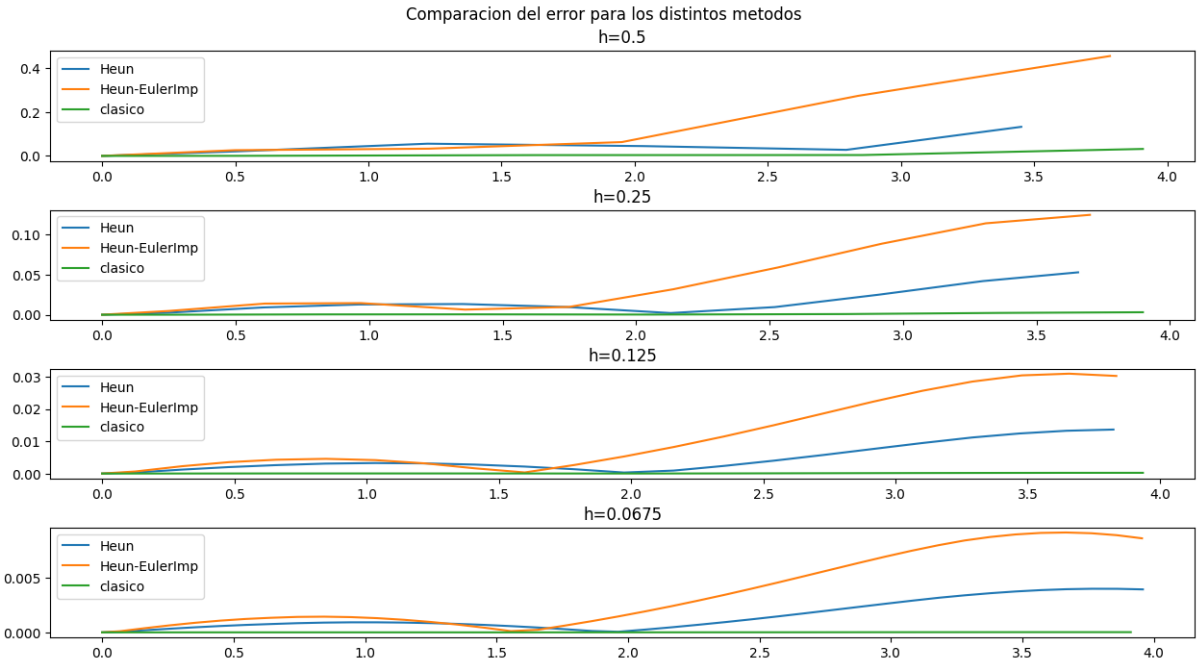

Figura 4.3: Errores cometidos en el problema del péndulo para distintos valores de h.

Observamos que, tal y como cabría esperar, lo errores cometidos por el método

clásico, que es de orden cuatro, son mucho más pequeños que los cometidos por los métodos de orden dos.

Para concluir con este problema, terminaremos por generar una animación del movimiento del péndulo aplicable tanto al caso general como al linealizado. Las animaciones con Python se hacen con el método FuncAnimation de la sublibrería animation del módulo matplotlib y se pueden guardar en varios formatos, incluidos GIF, MP4 y AVI. Para obtener información detallada sobre animaciones, puede consultarse su documentación oficial en https://matplotlib.org/stable/api/animation_api.html. Este es el código utilizado para generar la animación mencionada:

```
def anima_pendulo(t_0,t_f,y_0,metodo,problema):
    t,v=resuelve_problema(almacen(metodo),problema,t_0,t_f,
       y_0,0.01)
    y1=v[:,0]
    #parametrizamos la trayectoria del pendulo
    x=9.8*np.sin(y1)
    y=-9.8*np.cos(y1)
    fig=plt.figure()
    #definimos una funcion auxiliar para actualizar la
       posicion del cono en cada instante
    def actualizar(i):
        fig.clear()
        #dibujamos la circunferencia completa
        plt.plot(9.8*np.cos(t),9.8*np.sin(t))
        #dibujamos el punto de la circunferencia en el que
           esta el pendulo en el instante correspondiente
        plt.scatter(x[i], y[i], s=300,c="red")
        #dibujamos la varilla uniendo el punto anterior con
            el origen mediante una recta
        plt.plot([x[i],0],[y[i],0],color="black")
        #establecemos los ejes
        plt.xlim(-10.5,10.5)
        plt.ylim(-10.5,10.5)
    writervideo = animation.FFMpegWriter(fps=20)
    ani=animation.FuncAnimation(fig,actualizar,range(len(t)
       ),interval=1,repeat=False)
    ani.save('pendulo.mp4', writer=writervideo)
    return ani
```

Para el buen funcionamiento de este programa, es necesario tener instalado FFmpeg en el equipo, de lo contrario al ejecutar el código aparecería el siguiente error: *FileNotFoundError: [WinError 2] El sistema no puede encontrar el archivo especificado.*El método encargado de guardar los vídeos tiene dependencia con este programa. Sin embargo, al instalar el módulo matplotlib, no se incluye FFmpeg en caso de no estar ya instalado, lo que es un defecto del instalador.

Para instalar y configurar las variables de entorno correctamente en el sistema utilizado (Windows en este caso), se siguieron las instrucciones dada por esta página: https://es.101-help.com/f9ac8627f0-como-instalar-y-usar-ffmpeg-en-windows-10/. Además, como la página de descarga de FFmpeg no consta de una interfaz intuitiva para el usuario, se recomienda consultar el siguiente enlace para facilitar la descarga: https://browzwear.com/services/downloads/ffmpeg/#:~:text=Browzwear%20Help%20Center.-,Windows,files%20click%20the%20Windows%20icon.

Una vez configurado el equipo debidamente, generamos la animación mediante las siguientes instrucciones, según nos interese el caso linealizado o el general, respectivamente:

```
#anima_pendulo(0,2*np.pi,np.array([0,1]),"clasico",
    pendulo_linealizado)
#anima_pendulo(0,2.15*np.pi,np.array([0,1]),"clasico",
    pendulo)
```

Aquí mostramos algunos fotogramas pertenecientes a la animación generada:

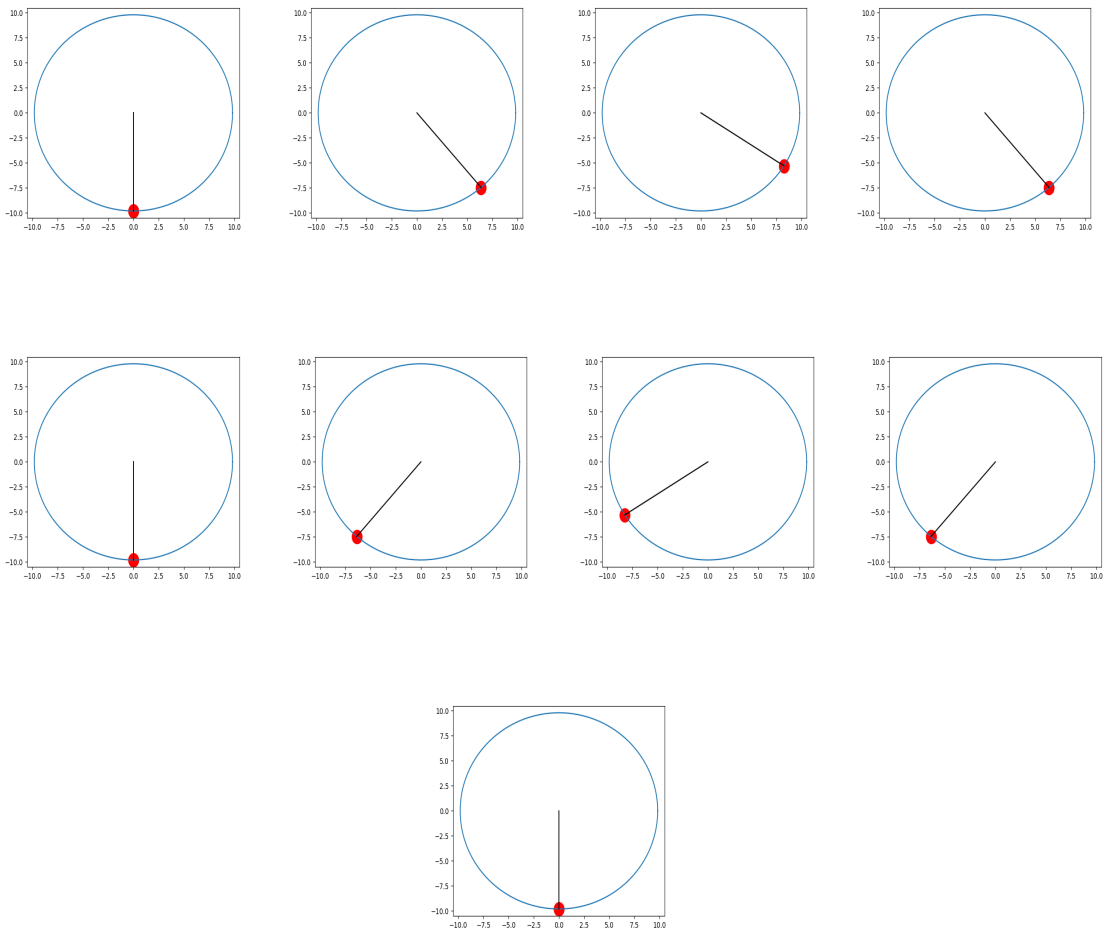

Figura 4.4: Fotogramas pertenecientes a la animación generada para el péndulo.

4.2.2. Test de la peonza

En este test nuestro objetivo es simular el movimiento de una peonza. Para ello, consideramos un tronco de cono descrito en coordenadas cilíndricas por los puntos de la forma

$$x = rs\cos\theta, \quad y = rs\sin\theta, \quad z = s - \frac{3a}{4}$$

$$r \in (0, b), \quad 0 < \theta < 2\pi, \quad 0 < s < a$$

siendo $b = \tan\varphi$ con $\varphi \in (0, \pi/2)$ la mitad del ángulo del cono.

Los puntos del cono se describen como:

$$u(t) + P(t) \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

con u el vector de posición del centro de masas y P una matriz de giro. El problema es calcular u y P. Suponemos que el vértice del cono se encuentra fijo, entonces tenemos

$$u(t) = \frac{3a}{4} \begin{pmatrix} P_{13} \\ P_{23} \\ P_{33} \end{pmatrix},$$

y usando la mecánica del sólido rígido, se tienen las ecuaciones para P

$$\begin{cases} \gamma' = 0 \\ \lambda' = \dfrac{4b^2-1}{4b^2+1}\gamma\mu - \dfrac{20g}{a(1+4b^2)}P_{31} \\ \mu' = -\dfrac{4b^2-1}{4b^2+1}\gamma\lambda - \dfrac{20g}{a(1+4b^2)}P_{32} \\ P'_{11} = -\gamma P_{12} - \lambda P_{13} \\ P'_{12} = \gamma P_{11} - \mu P_{13} \\ P'_{13} = \lambda P_{11} + \mu P_{12} \\ P'_{21} = -\gamma P_{22} - \lambda P_{23} \\ P'_{22} = \gamma P_{21} - \mu P_{23} \\ P'_{23} = \lambda P_{21} + \mu P_{22} \\ P'_{31} = -\gamma P_{32} - \lambda P_{33} \\ P'_{32} = \gamma P_{31} - \mu P_{33} \\ P'_{33} = \lambda P_{31} + \mu P_{32}. \end{cases}$$

El vector (γ, μ, λ) está relacionado con la velocidad ángular del sólido. Este sistema se almacena con este código:

```
def peonza(t,y):
    a=0.1
    phi=np.pi/10
    b=np.tan(phi)
    gamma=200
    g=9.8
    c1=(4*b**2-1)*gamma/(4*b**2+1)
    c2=20*g/(a*(1+4*b**2))
    return np.array([c1*y[1]-c2*y[8],
                     -c1*y[0]-c2*y[9],
                     -gamma*y[3]-y[0]*y[4],
                     gamma*y[2]-y[1]*y[4],
                     y[0]*y[2]+y[1]*y[3],
                     -gamma*y[6]-y[0]*y[7],
                     gamma*y[5]-y[1]*y[7],
                     y[0]*y[5]+y[1]*y[6],
                     -gamma*y[9]-y[0]*y[10],
                     gamma*y[8]-y[1]*y[10],
                     y[0]*y[8]+y[1]*y[9]])
```

Para la resolución, se ha elegido $a = 0{,}1$, $\varphi = \pi/10$, $\gamma_0 = 40$, $\lambda_0 = \mu_0 = 0$

$$P_0 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\zeta_0 & -\sin\zeta_0 \\ 0 & \sin\zeta_0 & \cos\zeta_0 \end{pmatrix},$$

con $\zeta_0 = \pi/40$ (este ángulo representa la inclinación inicial del cono con respecto al plano OXY). Aplicamos el método test_irresoluble previamente expuesto para obtener una estimación del error ejecutando el método entre 0 y 3 segundos, utilizando los métodos de Euler encajado con Heun, Bogacki-Shampine y el Runge-Kutta clásico.

```
ang=np.pi/40
#test_irresoluble(peonza,0,3,np.array([0,0,1,0,0,0,np.cos(
   ang),-np.sin(ang),0,np.sin(ang),np.cos(ang)]),
#["Heun","Bogacki-Shampine","clasico
   "],0.001,[0.1,0.05,0.025])
```

En la tabla 4.3 se presentan los errores obtenidos tomando como solución de referencia la obtenida mediante el método de Fehlberg:

Metodo	h	Error
Euler − Heun	0,1	1,1419875001245572
Euler − Heun	0,05	0,2967566908525976
Euler − Heun	0,025	0,07422711346855537
Bogacki − Shampine	0,1	0,6662590976805935
Bogacki − Shampine	0,05	0,08225387791507366
Bogacki − Shampine	0,025	0,010592900665676197
Clasico	0,1	0,005967137237251063
Clasico	0,05	0,0003736971051671589 3
Clasico	0,025	$2{,}3455705387753767e-05$

Cuadro 4.3: Errores cometidos para el problema de la peonza.

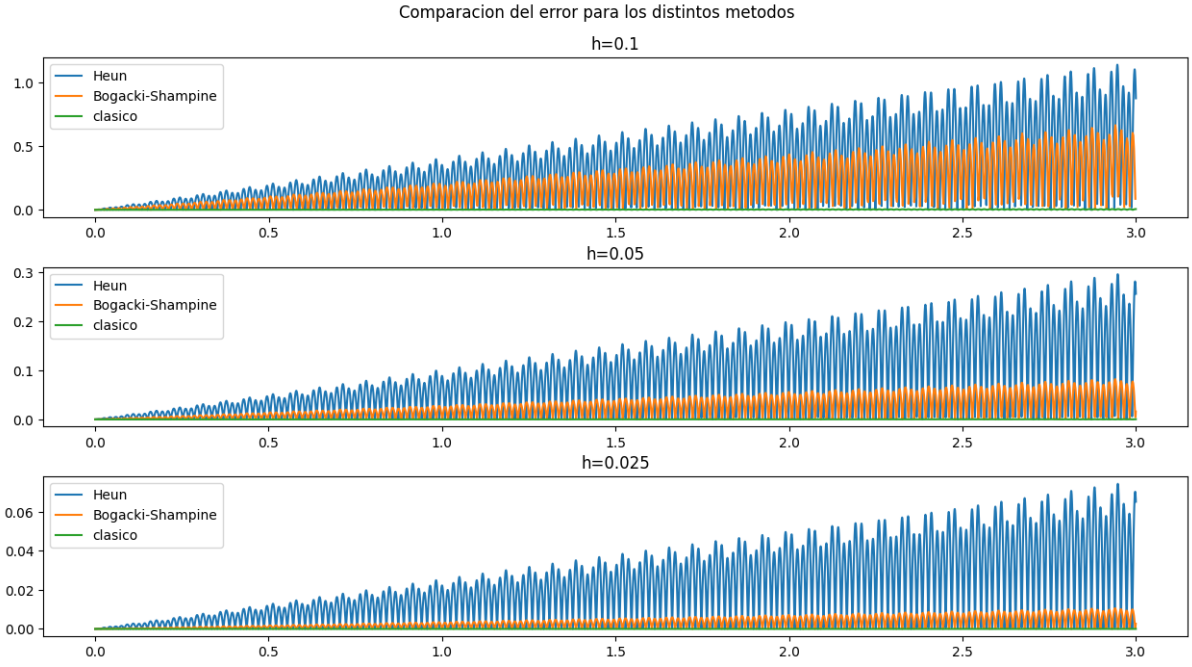

Figura 4.5: Errores cometidos en el problema de la peonza.

Como cabe esperar, los métodos tienen una mejor precisión en función de su orden de convergencia. Por ejemplo, para el método de orden 2, Euler-Heun, con $h = 0,025$, se obtiene un error de $7,42e^{-2}$; para obtener un error similar con el método de orden 3 (Bogacki-Shampine), de $8,22e^{-2}$, se ha necesitado la mitad de número de puntos.

A continuación, aplicando la parametrización que hemos explicado al inicio del problema, vamos a representar el movimiento de la peonza. Este es el código correspondiente:

```
def anima_peonza(angulo,t_0,t_f,y_0,metodo):
    t,v=resuelve_problema(almacen(metodo),peonza,t_0,t_f,
       y_0,0.01)
    interp = interpolate.interp1d(t, v[:,0], kind = "linear
       ")
    T=t_f/400
    j=0
    #almacenamos los instantes que vamos a representar
    indices=[]
    for i in range(0,400):
        tj=t_0+i*T
        while t[j]<=tj:
            j=j+1
        indices.append(j)
    #guardamos el vector correspondiente a cada incognita
       por separado
    p11=v[:,2]
    p12=v[:,3]
    p13=v[:,4]
    p21=v[:,5]
    p22=v[:,6]
    p23=v[:,7]
    p31=v[:,8]
    p32=v[:,9]
    p33=v[:,10]
    r=np.tan(angulo)
    s=np.linspace(0,0.1,50)
    #generamos el cono
    theta=np.linspace(0,2*np.pi,50)
    S,A=np.meshgrid(s,theta)
    xo=r*S*np.cos(A)
    yo=r*S*np.sin(A)
    zo=S-0.3/4
    fig=plt.figure()
    ax=fig.gca(projection='3d')
    #definimos una funcion auxiliar para actualizar la
       posicion del cono en cada instante
    def actualizar(i):
        ax.clear()
        u=0.075*np.array([p13[i],p23[i],p33[i]])
        X=u[0]+p11[i]*xo+p12[i]*yo+p13[i]*zo
        Y=u[1]+p21[i]*xo+p22[i]*yo+p23[i]*zo
        Z=u[2]+p31[i]*xo+p32[i]*yo+p33[i]*zo
        #representamos la superficie
        ax.plot_surface(X,Y,Z)
        #establecemos los ejes
```

```
        ax.set_xlim(-0.1, 0.1)
        ax.set_ylim(-0.1, 0.1)
        ax.set_zlim(0, 0.1)
    #generamos la animacion y la almacenamos en video
    writervideo = animation.FFMpegWriter(fps=20)
    anim=animation.FuncAnimation(fig,actualizar,indices,
       interval=1,repeat=False,blit=False)
    anim.save('peonza.mp4', writer=writervideo)
    return anim
```

Podemos observar que, al igual que en otros entornos, como matlab, la representación de superficies se basa en crear una cuadrícula con el rango de posibles valores de los parámetros de los que depende nuestra superficie y escribir la parametrización de la superficie en cuestión en función de la cuadrícula. Asimismo, para generar la superficie se utiliza el método *gca(projection='3d')*, que nos crea unos ejes tridimensionales vacíos. Para dibujar, ejecutamos el método *plot_surface* sobre dichos ejes tomando como argumento las coordenadas de los puntos de la superficie. La animación se ejecuta con la siguiente línea:

```
#anima_peonza(np.pi/10,0,10,np.array([0,0,1,0,0,0,np.cos(
   ang),-np.sin(ang),0,np.sin(ang),np.cos(ang)]),"clasico")
```

En la figura 4.6 mostramos una imagen correspondiente a un instante de la animación:

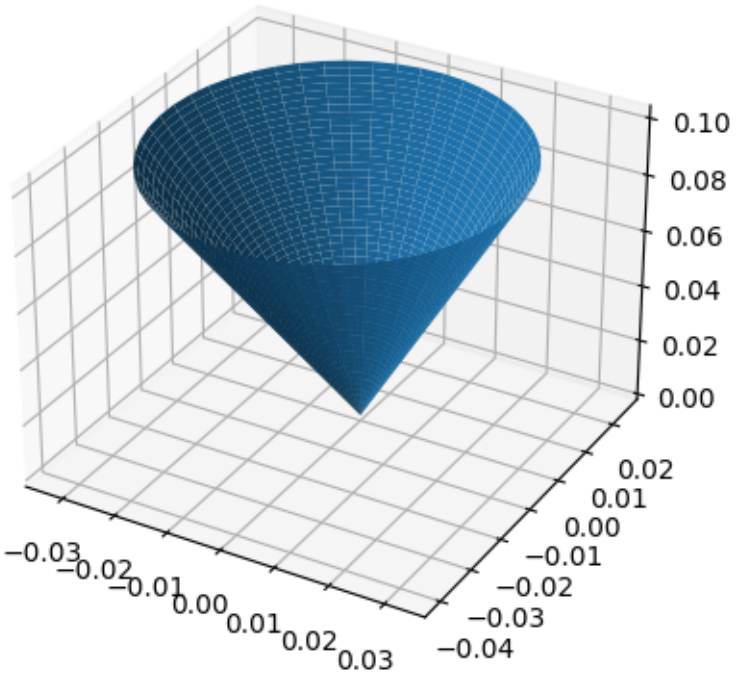

Figura 4.6: Movimiento de la peonza para un instante particular.

5. Análisis Numérico de Ecuaciones en Derivadas Parciales Elípticas

En este capítulo recordamos algunos de los resultados teóricos referentes a la existencia de solución de Ecuaciones en Derivadas Parciales Elípticas escritas en forma divergencia. Para ello, comenzamos recordando sin demostración algunas definiciones y resultados relativos a la teoría de distribuciones y a los espacios de Sobolev.

5.1. Complementos de Análisis Funcional

La mayoría de estos resultados aparecen en la asignatura Análisis Funcional y Ecuaciones en Derivadas Parciales.

Definición 5.1. *Dado un conjunto abierto $\Omega \subset \mathbb{R}^d$ definimos $\mathcal{D}(\Omega)$ como el espacio de las funciones de clase C^∞ en Ω con soporte compacto.*

Se dice que una sucesión φ_n converge en $\mathcal{D}(\Omega)$ hacia una función φ y se nota

$$\varphi_n \rightharpoonup \varphi \quad en\ \mathcal{D}(\Omega),$$

si el soporte de todos los elementos φ_n de la sucesión está contenido en un compacto de Ω y φ_n converge uniformemente hacia φ así como todas sus derivadas de cualquier orden.

Una distribución T es una función lineal de $D(\Omega)$ en $\mathbb{R}$ que es continua para la convergencia anterior, i.e.

$$\varphi_n \rightharpoonup \varphi \quad in\ \mathcal{D}(\Omega) \Rightarrow T\varphi_n \to T\varphi \quad en\ \mathbb{R}.$$

El espacio de distribuciones se denota por $\mathcal{D}'(\Omega)$ (espacio dual de $D(\Omega)$). Este espacio se suele dotar de la convergencia puntual (o convergencia $*$-débil)

$$T_n \rightharpoonup T \quad en\ \mathcal{D}'(\Omega) \iff \langle T_n, \varphi\rangle \to \langle T, \varphi\rangle, \quad \forall\, \varphi \in \mathcal{D}(\Omega).$$

Nos hemos estando refiriendo a Distribuciones reales, más generalmente, podemos tomar la distribuciones con valores en $\mathbb{C}$ en lugar de $\mathbb{R}$. La misma observación se tiene para los espacios que veremos a continuación.

Las distribuciones van a generalizar a las funciones. Más concretamente, consideremos $L^1_{loc}(\Omega)$ como el espacio de funciones medibles en Ω que son integrables en los

subconjuntos compactos de Ω. Cada función $f \in L^1_{loc}(\Omega)$ se puede identificar con la distribución $T_f : \mathcal{D}(\Omega) \to \mathbb{R}$ definida por

$$\langle T_f, \varphi \rangle = \int_\Omega f\varphi \, dx.$$

donde recordamos que la aplicación $f \in L^1_{loc}(\Omega) \to T_f \in \mathcal{D}(\Omega)$ es inyectiva debido al siguiente lema

Lema 5.2. Sea $f \in L^1_{loc}(\Omega)$ tal que

$$\int_\Omega f\varphi \, dx = 0, \quad \forall\, \varphi \in \mathcal{D}(\Omega)$$

entonces $f = 0$ e.c.t. Ω.

Definición 5.3. *Dada una distribución $T : \mathcal{D}(\Omega) \to \mathbb{R}$ y $1 \le i \le d$, se define la derivada parcial i-ésima de T, $\partial_i T \in \mathcal{D}(\Omega)$ por*

$$\langle \partial_i T, \varphi \rangle = -\langle T, \partial_i \varphi \rangle, \quad \forall\, \varphi \in \mathcal{D}(\Omega).$$

Por reiteración de esta fórmula se definen las derivadas de orden cualquiera. Además el operador derivada es continuo para la convergencia de distribuciones.

Esta definición junto con el hecho de que todas las funciones de $L^1_{loc}(\Omega)$ se identifican con distribuciones permite derivar infinitamente una función de $L^1_{loc}(\Omega)$. Esta derivada coincide con la usual para funciones regulares. Concretamente se tiene

Teorema 5.4. *Si u es una función $C^1(\Omega)$ entonces la derivada distribucional coincide con la derivada usual.*

Recordamos ahora las definiciones de los espacios $L^p(\Omega)$ y $W^{k,p}(\Omega)$ así como algunos resultados fundamentales.

Definición 5.5. *Para $1 \le p \le \infty$ y $\Omega \subset \mathbb{R}^d$ medible, se define $L^p(\Omega)$ por*

$$L^p(\Omega) = \Big\{ u : \Omega \to \mathbb{R} \text{ medible} : \quad \int_\Omega |u|^p dx < \infty \Big\}, \quad \text{si } 1 \le p < \infty,$$

$$L^\infty(\Omega) = \Big\{ u : \Omega \to \mathbb{R} \text{ medible} : \quad \exists\, M \ge 0,\ |u| \le M \ \text{ e.c.t. } \Omega \Big\}, \quad \text{si } p = \infty,$$

Proposición 5.6. Los espacios $L^p(\Omega)$ son Banach con las normas

$$\|u\|_{L^p(\Omega)} = \left(\int_\Omega |u|^p dx \right)^{\frac{1}{p}}, \quad \text{si } 1 \le p < \infty.$$

$$\|u\|_{L^\infty(\Omega)} = \min \Big\{ M \ge 0 : \quad |u(x)| \le M \ \text{ e.c.t. } \Omega \Big\}.$$

El espacio $L^2(\Omega)$ es un espacio de Hilbert con la norma

$$(u,v)_{L^2(\Omega)} = \int_\Omega uv\,dx, \quad \forall\, u,v \in L^2(\Omega).$$

Además, definiendo el conjugado de $p \in [1,\infty]$ por $p' = p/(p-1)$, se tienen las desigualdades de Young y Hölder

$$xy \le \frac{x^p}{p} + \frac{y^{p'}}{p'},\ \forall\, x,y>0, \qquad \int_\Omega |fg|\,dx \le \|f\|_{L^p(\Omega)}\|g\|_{L^{p'}(\Omega)},\ \forall\, f \in L^p(\Omega),\ g \in L^{p'}(\Omega).$$

Se tiene además el siguiente resultado de densidad y la siguiente caracerización del dual de funciones de $L^p(\Omega)$

Teorema 5.7. *El espacio $\mathcal{D}(\Omega)$ es denso en $L^p(\Omega)$ con $1 \le p < \infty$.*

Teorema 5.8. *Si $1 \le p < \infty$, el dual de $L^p(\Omega)$ se identifica con $L^{p'}(\Omega)$ mediante la aplicación $F : L^{p'}(\Omega) \to L^p(\Omega)$ definida por*

$$\langle Fu, v\rangle = \int_\Omega uv\,dx, \quad \forall\, u \in L^{p'}(\Omega),\ \forall\, v \in L^p(\Omega).$$

Definición 5.9. *Para $1 \le p \le \infty$ y $\Omega \subset \mathbb{R}^d$ abierto, se define el espacio $W^{1,p}(\Omega)$ por*

$$W^{1,p}(\Omega) = \left\{u \in L^p(\Omega),\ \nabla u \in L^p(\Omega)^d\right\}.$$

Este espacio es un espacio de Banach con la norma

$$\|u\|_{W^{1,p}(\Omega)} = \left(\|u\|^p_{L^p(\Omega)} + \|\nabla u\|^p_{L^p(\Omega)^d}\right)^{\frac{1}{p}},\ 1 \le p \le \infty.$$

En el caso particular $p = 2$, se suele notar el espacio $W^{1,2}(\Omega)$ por $H^1(\Omega)$. Este espacio es de Hilbert con el producto escalar

$$(u,v)_{H^1(\Omega)} = \int_\Omega \Big(uv + \nabla u \cdot \nabla v\Big)dx, \quad \forall\, u,v \in H^1(\Omega).$$

Análogamente se pueden definir los espacios $W^{k,p}(\Omega)$ como los espacios de funciones que son k veces derivables con derivadas en $L^p(\Omega)$.

Definición 5.10. *Para $1 \le p < \infty$, se define $W^{1,p}_0(\Omega)$ como el cierre de $D(\Omega)$ en $W^{1,p}(\Omega)$. Si Ω es acotado, una norma equivalente a la de $W^{1,p}(\Omega)$ en este espacio viene dada por*

$$\|u\|_{W^{1,p}(\Omega)} = \|\nabla u\|_{L^p(\Omega)^d}.$$

El espacio dual de $W^{1,p}_0(\Omega)$ es una espacio de distribuciones que se suele notar $W^{-1,p'}(\Omega)$.

Contrariamente a los espacios $L^p(\Omega)$ en los cuales $\mathcal{D}(\Omega)$ es denso, en general los espacios $W_0^{1,p}(\Omega)$ y $W^{1,p}(\Omega)$ son distintos (esencialmente sólo son iguales para $\Omega = \mathbb{R}^d$). Sí que se tiene sin embargo el siguiente resultado de densidad

Teorema 5.11. *Si Ω es un abierto regular de clase C^1, entonces $C^\infty(\overline{\Omega})$ es denso en $W^{1,p}(\Omega)$.*

Otra propiedad importante de los espacios de Sobolev es que la existencia de derivadas en $L^p(\Omega)$ hace que estas funciones estén en un espacio de funciones más regulares que $L^p(\Omega)$. Concretamente, se tiene

Teorema 5.12. *Sea Ω un conjunto abierto de $\mathbb{R}^d$.*

- *Si $1 \leq p < d$ ($p = d$ si $d = 1$) el espacio $W_0^{1,p}(\Omega)$ se inyecta con inyección continua en $L^{\frac{Np}{d-p}}(\Omega)$, y si Ω es acotado con inyección compacta en $L^r(\Omega)$, $1 \leq r < Np/(d-p)$.*
- *Si $p > d$ y Ω es acotado $W_0^{1,p}(\Omega)$ se inyecta con inyección compacta en $C^0(\overline{\Omega})$.*

Los resultados anteriores son ciertos cambiando $W_0^{1,p}(\Omega)$ por $W^{1,p}(\Omega)$ si Ω es acotado y su frontera es localmente el grafo de una función Lipschitziana.

Se suele entender el espacio $W_0^{1,p}(\Omega)$ como el espacio de funciones de $W^{1,p}(\Omega)$ que se anulan en la frontera. Un resultado en ese sentido viene dado por el siguiente teorema:

Teorema 5.13. *Sea Ω un conjunto abierto y acotado de $\mathbb{R}^d$ de clase C^1, $1 \leq p < d$, entonces existe una aplicación $\mathcal{T} : W^{1,p}(\Omega) \to L^{\frac{(d-1)p}{d-p}}(\partial\Omega)$ lineal y continua tal que*

$$\mathcal{T}u = u|_{\partial\Omega}, \ \forall\, u \in C^1(\overline{\Omega}).$$

El resultado anterior permite definir el valor en la frontera de una función de $W^{1,p}(\Omega)$ si $p < d$ como el valor de $\mathcal{T}u$. Teniendo en cuenta que por el Teorema 5.12, $W^{1,d}(\Omega)$ se inyecta en $W^{1,q}(\Omega)$ para todo $q < d$ tenemos que la traza de una función en $W^{1,d}(\Omega)$ está en $L^r(\Omega)$ para todo $r < \infty$. En el caso $p > d$, sabemos que las funciones de $W^{1,p}(\Omega)$ son continuas en $\bar{\Omega}$ y por tanto están definidas también en la frontera como funciones continuas.

Observación 5.14. El operador $\mathcal{T}$ definido en el teorema anterior (y el correspondiente si $p \geq d$) no es sobreyectivo. Su imagen se suele denotar por $W^{\frac{1}{p'},p}(\partial\Omega)$ ($H^{\frac{1}{2}}(\Omega)$ si $p = 2$). Se trata de un espacio de Banach (Hilbert si $p = 2$) con la norma

$$\|w\|_{W^{\frac{1}{p'},p}(\partial\Omega)} = \text{ínf}\left\{\|u\|_{W^{1,p}(\Omega)} : \ u = w \ \text{ sobre } \partial\Omega\right\}.$$

Para $1 \leq p < \infty$, el dual de este espacio se denota $W^{-\frac{1}{p'},p'}(\Omega)$. ($H^{-\frac{1}{2}}(\Omega)$, si $p = 2$).

Además del Teorema anterior recordamos también otro teorema de trazas que usaremos para resolver ecuaciones de primer orden. Este resultado se tiene en el espacio $L^p(\Omega; \text{div})$ que definimos a continuación.

Definición 5.15. *Para $1 \leq p \leq \infty$, se define el espacio $L^p(\Omega;\mathrm{div})$ por*

$$L^p(\Omega;\mathrm{div}) = \left\{ f \in L^p(\Omega)^d : \quad \mathrm{div} f \in L^p(\Omega) \right\},$$

el cual es un espacio de Banach con la norma

$$\|f\|_{L^p(\Omega;\mathrm{div})} = \left(\|f\|^p_{L^p(\Omega)^d} + \|\mathrm{div} f\|^p_{L^p(\Omega)} \right)^{\frac{1}{p}}.$$

Cuando $p = 2$, el espacio es de Hilbert.

Teorema 5.16. *Si $\Omega \in C^1$ y $1 \leq p < \infty$, el espacio $C^\infty(\overline{\Omega})$ es denso en $L^p(\Omega;\mathrm{div})$.*

Teorema 5.17. *Sea Ω un conjunto abierto acotado de clase C^1, entonces para $1 < p \leq \infty$, existe un operador $\mathcal{T} : L^p(\Omega;\mathrm{div}) \to W^{-\frac{1}{p},p}(\partial\Omega)$ lineal y continuo tal que*

$$\mathcal{T}(f) = f \cdot \nu, \quad \forall\, f \in C^\infty(\overline{\Omega}),$$

siendo ν el vector normal unitario exterior a Ω sobre $\partial\Omega$.

Se tiene además la siguiente extensión del teorema de la divergencia

$$\int_\Omega f \cdot \nabla v \, dx = \langle \mathcal{T}(f), v \rangle - \int_\Omega \mathrm{div}\, f \, v \, dx, \quad \forall\, f \in L^p(\Omega;\mathrm{div}),\ \forall\, v \in W^{1,p}(\Omega).$$

El resultado anterior permite definir el valor del producto escalar $f \cdot \nu$ sobre $\partial\Omega$ para una función $f \in L^p(\Omega;\mathrm{div})$, el cual identificaremos con el de $\mathcal{T} f$ y que se conoce como traza normal de f.

5.2. Formulación variacional de problemas lineales elípticos.

En esta sección recordamos algunos resultados relativos a la existencia y unicidad de solución de un problema de EDP elíptico, escrito en forma divergencia. Las soluciones se entenderán en el sentido débil o distribucional, para lo cual es necesario introducir lo que se conoce como formulación variacional del problema.

Sea Ω un conjunto abierto y acotado de $\mathbb{R}^d$. Dadas $A \in L^\infty(\Omega)^{d\times d}$, $b, c \in L^\infty(\Omega)^d$ y $d \in L^\infty(\Omega)$, estamos interesados en el operador diferencial

$$\mathcal{L}u = -\mathrm{div}(A\nabla u + bu) + c \cdot \nabla u + du, \tag{5.1}$$

donde a la función A le pediremos usualmente la propiedad de elipticidad siguiente

$$\exists \alpha > 0 \ \text{ tal que } \ (A(x)\xi) \cdot \xi \geq \alpha |\xi|^2, \quad \forall\, \xi \in \mathbb{R}^d, \ \text{ e.c.t. } x \in \Omega. \tag{5.2}$$

Nuestro objetivo es estudiar problemas de EDP relativos a este operador, en los cuales recordamos es necesario tener en cuenta no solo la ecuación, sino también las condiciones de contorno.

Dadas $f \in L^2(\Omega)$ y $w \in H^1(\Omega)$, el problema de Dirichlet correspondiente al operador $\mathcal{L}$, se escribe

$$\begin{cases} \mathcal{L}u = f & \text{en } \Omega \\ u = w & \text{sobre } \partial\Omega. \end{cases} \tag{5.3}$$

En este problema vamos a entender que la ecuación se verifica en el sentido de las distribuciones. La función incógnita u la buscaremos en el espacio $H^1(\Omega)$. Teniendo en cuenta la definición de derivada distribucional y la densidad de $D(\Omega)$ en $H_0^1(\Omega)$, se tiene entonces que u es solución de (5.3) si y sólo si, se verifica

$$\begin{cases} u - w \in H_0^1(\Omega) \\ \mathfrak{a}(u,v) = \displaystyle\int_\Omega fv\,dx, \quad \forall\, v \in H_0^1(\Omega), \end{cases} \tag{5.4}$$

donde $\mathfrak{a} : H^1(\Omega) \times H^1(\Omega) \to \mathbb{R}$ es la forma bilineal y continua definida por

$$\mathfrak{a}(u,v) = \int_\Omega \Big((A\nabla u + bu) \cdot \nabla v + c \cdot \nabla u\, v + duv \Big) dx, \quad \forall\, u, v \in H^1(\Omega). \tag{5.5}$$

La formulación (5.4) es lo que se conoce como formulación variacional del problema (5.3). La función v que aparece en la formulación se denomina función test. La existencia de solución de este tipo de problemas se suele llevar a cabo usando el teorema de Lax-Milgram que recordamos a continuación.

Teorema 5.18. ***(Teorema de Lax-Milgram)*** *Sea H un espacio de Hilbert, $\mathfrak{a} : H \times H \to \mathbb{R}$ una forma bilineal, continua y coercitiva, en el sentido de que existe $\alpha > 0$ tal que*

$$\mathfrak{a}(u,u) \geq \alpha \|u\|^2, \quad \forall\, u \in H. \tag{5.6}$$

Entonces, para toda $f \in H'$, existe una única $u \in H$ solución de

$$\mathfrak{a}(u,v) = \langle f, v \rangle, \quad \forall\, v \in H. \tag{5.7}$$

Además, si $\mathfrak{a}$ es simétrica, entonces u está caracterizada como la única solución del problema de optimización

$$\min_{v \in H} \Big\{ \frac{1}{2}\mathfrak{a}(v,v) - \langle f, v \rangle \Big\}. \tag{5.8}$$

Con este resultado y teniendo en cuenta la desigualdad de Poincaré en el espacio $H_0^1(\Omega)$, se puede probar por ejemplo el siguiente teorema de existencia y unicidad para el problema (5.3). Recordamos que una distribución T en Ω se dice no negativa si verifica

$$\langle T, \varphi \rangle \geq 0, \quad \forall\, \varphi \in \mathcal{D}(\Omega),\ \varphi \geq 0 \ \text{ en } \Omega.$$

Teorema 5.19. *Sea Ω un conjunto abierto y acotado de $\mathbb{R}^d$, $A \in L^\infty(\Omega)^{d\times d}$, $b, c \in L^\infty(\Omega)^d$ y $d \in L^\infty(\Omega)$ tales que se verifica (5.2) y*

$$-\operatorname{div}(b+c) + d \geq 0 \quad \text{en } \Omega,$$

entonces, para toda $f \in L^2(\Omega)$ y toda $w \in H^1(\Omega)$, existe una única solución $u \in H^1(\Omega)$ del problema (5.3).

La condición de contorno en (5.3) corresponde a las conocidas como condiciones de tipo Dirichlet en $\partial\Omega$. Es habitual también considerar el problema

$$\begin{cases} \mathcal{L}u = f \quad \text{en } \Omega \\ (A\nabla u + bu)\cdot\nu + \kappa u = g \quad \text{sobre } \partial\Omega. \end{cases} \tag{5.9}$$

donde ν es el vector normal unitario exterior sobre $\partial\Omega$, $\kappa \in L^\infty(\partial\Omega)$ y $g \in L^2(\partial\Omega)$. Esto es lo que conoce como una condición de contorno de tipo Robin o Fourier, que en el caso $\kappa = 0$ se llama condición de Neumann. A diferencia del caso de Dirichlet, necesitamos cierta regularidad sobre la frontera de forma que el operador traza esté bien definido.

Buscando como antes la solución en el espacio $H^1(\Omega)$, entendiendo que la ecuación se verifica en el sentido de las distribuciones y recordando la existencia de una traza normal para funciones de L^2 con gradiente en L^2, se tiene que decir que u es solución de (5.9), es equivalente a decir que u verifica la formulación variacional o formulación débil

$$\begin{cases} \text{Hallar } u \in H^1(\Omega) \\ \mathfrak{a}(u,v) = \displaystyle\int_\Omega fv\,dx + \int_{\partial\Omega} gv ds(x), \quad \forall\, v \in H^1(\Omega), \end{cases} \tag{5.10}$$

donde ahora la aplicación bilineal $\mathfrak{a} : H^1(\Omega)\times H^1(\Omega) \to \mathbb{R}$ está definida por

$$\mathfrak{a}(u,v) = \int_\Omega \Big((A\nabla u + bu)\cdot\nabla v + c\cdot\nabla u\, v + duv \Big) dx + \int_{\partial\Omega} \kappa uv\, ds(x), \quad \forall\, u, v \in H^1(\Omega). \tag{5.11}$$

Usando el teorema de Lax-Milgram podemos por ejemplo probar el resultado siguiente

Teorema 5.20. *Sea Ω un conjunto abierto y acotado de $\mathbb{R}^d$ de clase C^1, $A \in L^\infty(\Omega)^{d\times d}$, $b, c \in L^\infty(\Omega)^d$, $d \in L^\infty(\Omega)$ y $\kappa \in L^\infty(\partial\Omega)$ tales que se verifica (5.2) junto a*

$$-\operatorname{div}(b+c) + 2d \geq 0 \quad \text{en } \Omega, \qquad (b+c)\cdot\nu + 2\kappa \geq 0 \quad \text{sobre } \partial\Omega$$

$-\operatorname{div}(b+c) + 2d$ *no identicamente nula o* $(b+c)\cdot\nu + 2\kappa$ *no identicamente nula.*

Entonces, para toda $f \in L^2(\Omega)$ y toda $g \in L^2(\partial\Omega)$, existe una única solución $u \in H^1(\Omega)$ del problema (5.9).

Observación 5.21. En el teorema anterior hemos tomado la frontera de clase C^1, para que tengan sentido los operadores traza. Más generalmente se sabe que estos operadores están bien definidos en el caso de abiertos cuya frontera es Lipschitz, esto incluye por ejemplo el caso de un abierto poliédrico que será el caso que consideraremos usualmente en la resolución numérica de este tipo de problemas.

Observación 5.22. Más generalmente que los problemas (5.3) y (5.9), se pueden también considerar problemas mixtos donde se impone una condición de tipo Dirichlet en una parte de $\partial\Omega$ y una condición de tipo Neumann en el resto de $\partial\Omega$. Las funciones test en este caso pertenecen a $H^1(\Omega)$ y se anulan sobre la frontera donde se ha impuesto la condición de tipo Dirichlet.

Observación 5.23. Los Teoremas 5.19 y 5.20 proporcionan la existencia de soluciones en el espacio $H^1(\Omega)$. Una cuestión importante es si estas funciones son en realidad más regulares, así por ejemplo nos podemos preguntar si se encuentran en un espacio de tipo $W^{k,p}(\Omega)$, lo que gracias a los resultados de inyección de Sobolev, y suponiendo los coeficientes suficientemente regulares, implicará en particular que si k, p son suficientemente grandes, entonces u es solución clásica, i.e. $u \in C^2(\overline{\Omega})$. Bastaría de hecho $k \geq 3$, $p > d$. También se pueden buscar directamente soluciones en los espacios de funciones Hölderianas en lugar de espacios de Sobolev. Se trata de un tipo de resultados bastante complejos que escapan del presente trabajo. Citamos por ejemplo los capítulos 4, 8 y 9 de [6]. En relación con las estimaciones de error que obtendremos más adelante en la resolución numérica de estos problemas, enunciamos el siguiente resultado relativo al problema (5.3) (ver los teoremas 0.12 y 8.13 de [6]). Un resultado similar es también cierto para (5.9)

Teorema 5.24. *En las condiciones del teorema 5.19, supongamos $k \geq 2$, Ω de clase C^k, $A \in W^{k-1,\infty}(\Omega)^{d\times d}$, $d-\operatorname{div} b, c-b \in W^{k-2,\infty}(\Omega)$, $w \in H^k(\Omega)$, $f \in H^{k-2}(\Omega)$. Entonces la solución u de (5.3) pertenece a $H^k(\Omega)$.*

Observación 5.25. El resultado anterior implica en particular que si todos los datos del problema (5.3) son de clase C^∞, entonces la solución u pertenece a $C^\infty(\overline{\Omega})$.

Observación 5.26. Como dijimos antes, vamos a estar interesados en abiertos poliédricos, los cuales no son ni siquiera C^1. En este caso, se sabe que el teorema 5.24 sigue siendo cierto para $k = 2$, añadiendo la hipótesis Ω convexo. Ver e.g. [7], Teorema 3.2.1.2.

5.3. Aproximación de Galerkin de un problema variacional.

Comenzamos con esta sección el estudio numérico de un problema de tipo Lax-Milgram (ver Teorema 5.18). Aquí vamos a tratar la aproximación de un problema abstracto.

Sean H un espacio de Hilbert y $\mathfrak{a} : H \times H \to \mathbb{R}$ una forma bilineal, continua y coercitiva. Dada $f \in H'$, nuestro objetivo es aproximar la solución u del problema (5.7). El método de Galerkin consiste simplemente en sustituir el espacio H por un subespacio finito-dimensional $\hat{H} \subset H$ y aproximar u por la única solución $\hat{u}$ del problema

$$\begin{cases} \hat{u} \in \hat{H} \\ \mathfrak{a}(\hat{u}, v) = \langle f, v \rangle, \quad \forall\, v \in \hat{H}. \end{cases} \tag{5.12}$$

Este problema posee una única solución por el Teorema de Lax-Milgram, ya que al ser $\hat{H}$ un subespacio vectorial de dimensión finita, es también un espacio de Hilbert.

Usualmente, en lugar de un subespacio, lo que se hace es considerar una sucesión de subespacios dependiendo de un parámetro $h > 0$ y estudiar la convergencia cuando este parámetro tiende a cero. Sin embargo, por el momento preferimos describir el método con un solo espacio $\hat{H}$ para simplificar la notación.

La resolución efectiva de este problema, es como sigue: sea d la dimensión de $\hat{H}$ y $\{\phi_1, ..., \phi_d\}$ una base de $\hat{H}$. Por linealidad tenemos entonces que $\hat{u}$ es solución de (5.12) si y sólo si verifica

$$\begin{cases} \hat{u} \in \hat{H} \\ \mathfrak{a}(\hat{u}, \phi_i) = \langle f, \phi_i \rangle, \quad \forall\, i \in \{1, ..., d\}, \end{cases} \tag{5.13}$$

de forma que lo que tenemos son d ecuaciones para $\hat{u}$. Por otra parte, como $\hat{u}$ pertenece a $\hat{H}$, deben existir $\alpha_1, ..., \alpha_d \in \mathbb{R}$ tales que

$$\hat{u} = \sum_{j=1}^{d} \alpha_j \phi_j.$$

El problema se reduce entonces a encontrar estos α_j. Sustituyendo en (5.13) tenemos el sistema lineal de d ecuaciones con d incógnitas dado por

$$\sum_{j=1}^{d} \mathfrak{a}(\phi_j, \phi_i)\alpha_j = \langle f, \phi_i \rangle, \quad 1 \leq i \leq d.$$

Definiendo por tanto la matriz $\hat{A} \in \mathbb{R}^{d \times d}$ por

$$(\hat{A})_{ij} = \mathfrak{a}(\phi_j, \phi_i), \quad 1 \leq i, j \leq d, \tag{5.14}$$

y el vector $b \in \mathbb{R}^d$ por

$$b_i = \langle f, \phi_i \rangle, \quad 1 \leq i \leq d, \tag{5.15}$$

tenemos que resolver el problema (5.12) se reduce a resolver el sistema

$$\hat{A}\alpha = b, \tag{5.16}$$

con $\alpha = (\alpha_1, ..., \alpha_d)^t$. Es importante observar que el hecho de que $\mathfrak{a}$ verifica (5.6) implica que la matriz $\hat{A}$ verifica

$$(\hat{A}\xi) \cdot \xi \geq \alpha \left\| \sum_{i=1}^{d} \xi_i \phi_i \right\|^2, \quad \forall\, \xi = (\xi_1, ..., \xi_d)^t \in \mathbb{R}^d$$

y por tanto que $\hat{A}$ es definida positiva, propiedad de gran importancia a la hora de la resolución efectiva del sistema (5.16).

Una estimación del error cometido al aproximar la solución u de (5.7) por la solución $\hat{u}$ de (5.13) está dada por el siguiente teorema

Teorema 5.27. *Sean H un espacio de Hilbert, $\mathfrak{a} : H \times H \to \mathbb{R}$ una forma bilineal, continua, verificando (5.6) para algún $\alpha > 0$ y $f \in H'$, entonces para todo subespacio cerrado $\hat{H} \subset H$ se tiene que las soluciones u y $\hat{u}$ de los problemas (5.7) y (5.12) verifican*

$$\|u - \hat{u}\| \leq \frac{\|\mathfrak{a}\|}{\alpha}\|u - \pi_{\hat{H}}u\|, \tag{5.17}$$

con $\pi_{\hat{H}}u$ la proyección ortogonal de u sobre $\hat{H}$.

Demostración. La existencia de $\pi_{\hat{H}}u$ está garantizada por el hecho de que $\hat{H}$ es cerrado junto con el teorema de la proyección en espacios de Hilbert. Tomando $v = \pi_{\hat{H}}u - \hat{u} \in \hat{H} \subset H$ como función test tanto en (5.7) como en (5.12) y restando las correspondientes igualdades, tenemos

$$\mathfrak{a}(u - \hat{u}, \pi_{\hat{H}}u - \hat{u}) = 0,$$

que gracias a la bilinealidad de $\mathfrak{a}$ se escribe también como

$$\mathfrak{a}(u - \hat{u}, u - \hat{u}) = \mathfrak{a}(u - \hat{u}, u - \pi_{\hat{H}}u).$$

La coercitividad (5.6) y la continuidad de $\mathfrak{a}$ proporcionan entonces

$$\alpha\|u - \hat{u}\|^2 \leq \|\mathfrak{a}\|\|u - \hat{u}\|\|u - \pi_{\hat{H}}u\|$$

y por tanto (5.17). ∎

Observación 5.28. Como todo subespacio finito-dimensional de un espacio normado es cerrado, el Teorema 5.27 se aplica en particular a la aproximación mencionada anteriormente, si bien la condición de que $\hat{H}$ sea finito-dimensional no es necesaria para el teorema.

Observación 5.29. Como mencionamos anteriormente, es usual tomar no un solo espacio $\hat{H}$ sino una familia de espacios $\hat{H}_h$ dependientes de un parámetro h que tiende a cero. Si los espacios $\hat{H}_h$ son construidos de tal forma que para todo $u \in H$ existe una sucesión $u_h \in \hat{H}_h$ tal que u_h converge a u cuando h tiende a cero, la estimación (5.17) prueba entonces

$$\|u - \hat{u}_h\| \leq \frac{\|\mathfrak{a}\|}{\alpha}\|u - \pi_{\hat{H}_h}u\| \leq \frac{\|\mathfrak{a}\|}{\alpha}\|u - u_h\|,$$

con $\hat{u}_h$ la solución de (5.12) correspondiente a $\hat{H} = \hat{H}_h$. Esto prueba la convergencia del método de Galerkin y además proporciona una estimación del error, suponiendo que tenemos una cota efectiva de la diferencia $\|u - u_h\|$.

El problema que se plantea ahora es construir una sucesión efectiva de espacios $\hat{H}_h$. Esta es la cuestión que abordamos en las siguientes secciones para el caso $H = H^1(\Omega)$ que es el espacio que nos interesa en el estudio de problemas elípticos de EDP y que constituirá lo que se conoce como método de los elementos finitos.

En este caso, los espacios serán elegidos de tal forma que las bases correspondientes de los espacios $\hat{H}_h$ estarán formadas por funciones con soporte pequeño. Ello implicará que la correspondiente matriz $\hat{A}_h$ dada por (5.14) sea hueca, es decir que una amplia mayoría de las entradas de esta matriz esté formada por ceros, condición necesaria para que los sistemas lineales (5.16) puedan ser resueltos de forma efectiva ya que se tratará de sistemas con una cantidad enorme de incógnitas cuya resolución llevaría a una cantidad ingente de cálculos en el caso de matrices llenas. El método además tendrá la ventaja de que las aproximaciones u_h, mencionadas anteriormente, se podrán construir fácilmente mediante técnicas de interpolación.

5.4. Elementos finitos

En esta sección vamos a describir los elementos finitos de Lagrange, centrándonos en los de tipo simplicial y los paralelotopos. Como ya dijimos en la sección anterior, esto nos permitirá aproximar el espacio H^1 por un espacio finito dimensional, que será donde resolvamos los distintos problemas de EDPs.

5.4.1. Elementos finitos de Lagrange

Consideremos:

1. $K \subset \mathbb{R}^d$ compacto, conexo y de interior no vacío,
2. un espacio vectorial P de dimensión finita formado por funciones definidas de K en $\mathbb{R}$ y
3. $\Sigma = \{\varphi_1, \ldots, \varphi_n\}$ un conjunto de funciones lineales definidas sobre P.

Introducimos las siguientes definiciones:

Definición 5.30. *Diremos que el conjunto Σ es P-unisolvente si, dados n escalares reales cualesquiera α_j, $1 \leq j \leq n$, existe una única función p del espacio P tal que*

$$\varphi_j(p) = \alpha_j, \quad 1 \leq j \leq n. \tag{5.18}$$

Cuando el conjunto Σ es P-unisolvente, la tupla (K, P, Σ) se llama elemento finito. En el caso particular en que el espacio P corresponde a un espacio de polinomios de un determinado grado, se llama eslemento finito de Lagrange.

Ejemplo 5.31. Como caso más simple y que volveremos a ver más tarde en un marco más general, podemos tomar el espacio $\mathbb{P}_1$, que corresponde a

$$K = \Big\{(x_1, \ldots, x_d) : \ x_1, \ldots, x_d \geq 0, \ \sum_{i=1}^{n} x_i \leq 1\Big\},$$

$$P = \Big\{p : K \to \mathbb{R} : \ p \text{ polinomio de grado } \leq 1\Big\},$$

y denotando $z_1, \ldots, z_{d+1}$ los vértices de K,

$$\Sigma = \{\varphi_1, \ldots, \varphi_{d+1} \text{ con } \varphi_i(p) = p(z_i)\}.$$

En todos los ejemplos que contemplaremos en la presente memoria, las funciones φ_i siempre serán de la forma $\varphi_i(p) = p(a_i)$, con $a_i \in K$. En este caso, podremos realizar la identificación $\Sigma \approx \{a_i\}_{i=1}^n$

Dado un elemento finito (K, P, Σ), existe entonces para todo entero i, $1 \leq i \leq n$, una única función $p_i \in P$ tal que

$$p_i(a_j) = \delta_{ij}, \quad 1 \leq j \leq n. \tag{5.19}$$

Más generalmente para toda función $v : K \to \mathbb{R}$, existe una única $p \in P$ que interpola v sobre Σ, es decir satisfaciendo

$$p(a_j) = v(a_j), \quad 1 \leq j \leq n. \tag{5.20}$$

Definición 5.32. *Dado un elemento finito de Lagrange (K, P, Σ), denominamos funciones de base a las n funciones p_i, $1 \leq j \leq n$, definidas por (5.19). Llamaremos operador de p-interpolación de Lagrange sobre Σ al operador que a toda función v definida sobre K le asocia la función Πv que se define como*

$$\Pi v = \sum_{i=1}^{d} v(a_i) p_i, \tag{5.21}$$

y a Πv nos referiremos como la P-interpolada de Lagrange de v sobre Σ.

Observación 5.33. Una condición necesaria evidente para que el conjunto Σ sea P-unisolvente es que

$$dim(P) = card(\Sigma) = d \tag{5.22}$$

Además, también conviene observar que probar la P-unisolvencia equivale a probar la biyectividad de la aplicación $\mathcal{L} : P \to \mathbb{R}^d$ definida como:

$$\mathcal{L}(p) = (p(a_j))_{j=1}^d$$

que, al ser P d-dimensional, basta con probar que es inyectiva.

Vamos a ver ahora un método sistemático para generar toda una familia de elementos finitos a partir de un elemento finito (K, P, Σ). Para ello, sea $\hat{K}$ un compacto de $\mathbb{R}^d$, conexo con interior no vacío y sea $F : \hat{K} \to \mathbb{R}^d$. Supongamos que

$$K = F(\hat{K}) \tag{5.23}$$

es compacto, conexo y de interior no vacío (si F es una biyección bicontinua de $\hat{K}$ en K, esto se tiene gracias a las hipótesis impuestas sobre $\hat{K}$).

Teorema 5.34. *Supongamos que la aplicación F es inyectiva. Entonces, si $(\hat{K}, \hat{P}, \hat{\Sigma})$ es un elemento finito de Lagrange, la tupla (K, P, Σ), donde K viene dado por (5.23) y donde P y Σ verifican*

$$\begin{aligned} P &= \{p : K \to \mathbb{R};\ p \circ F \in \hat{P}\}, \\ \Sigma &= F(\hat{\Sigma}), \end{aligned} \tag{5.24}$$

es un elemento finito de Lagrange.

Este resultado nos conduce a la siguiente definición:

Definición 5.35. *Dos elementos finitos de Lagrange $(\hat{K}, \hat{P}, \hat{\Sigma})$ y (K, P, Σ) se dicen equivalentes si existe una biyección $F : \hat{K} \to K$ verificando (5.24). Además, cuando podamos elegir F como una aplicación afín invertible, los elementos finitos se dirán afín-equivalentes.*

Se tiene

Teorema 5.36. *Sean $(\hat{K}, \hat{P}, \hat{\Sigma})$ y (K, P, Σ) dos elementos finitos de Lagrange equivalentes y $F : \hat{K} \to K$ verificando (5.24). Si $\hat{\Pi}$ es el operador de $\hat{P}$-interpolación sobre $\hat{\Sigma}$, el operador Π de P-interpolación sobre Σ se caracteriza por*

$$(\Pi v) \circ F = \hat{\Pi}(v \circ F) \tag{5.25}$$

para toda función v definida sobre K.

La prueba de estos resultados puede encontrarse en la sección 4.1 de [5].

5.4.2. Elementos finitos simpliciales

Describimos en este apartado los elementos finitos más usuales, que serán los que utilizaremos en los experimentos numéricos. Comenzamos definiendo el concepto de $d-$símplex:

Definición 5.37. *Consideremos $(d+1)$ puntos $a_j = (a_{ij})_{i=1}^d \in \mathbb{R}^d$, $1 \leq j \leq d+1$ y supongamos que la matriz*

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1,d+1} \\ a_{21} & a_{22} & \dots & a_{2,d+1} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{d,d+1} \\ 1 & 1 & \dots & 1 \end{pmatrix} \tag{5.26}$$

Llamaremos d-símplex o símplice K de vértices a_j, $1 \leq j \leq d+1$ a la envolvente convexa de los puntos a_j.

Observación 5.38. Para $d = 2$, el símplice correspondiente es el triángulo; para $d = 3$, el tetraedro.

Vamos a construir una clase de elementos finitos de Lagrange (K, P, Σ) donde K será un d-símplex cualquiera de $\mathbb{R}^d$.

Definición 5.39. *Todo punto $x \in \mathbb{R}^d$ de coordenadas x_i, $1 \leq i \leq d$, se caracteriza por $(d+1)$ $\lambda_j = \lambda_j(x)$ definidos como solución del sistema lineal*

$$\begin{cases} \displaystyle\sum_{j=1}^{d+1} a_{ij}\lambda_j = x_i, \quad 1 \leq i \leq d, \\ \displaystyle\sum_{j=1}^{d+1} \lambda_j = 1, \end{cases} \tag{5.27}$$

donde los a_{ij} son los dados en la Definición 5.26.

A los escalares λ_j se les denomina coordenadas baricéntricas del punto x respecto a los a_j, $1 \leq j \leq d$. Las funciones $x = (x_i)_{1 \leq i \leq d} \mapsto \lambda_j(x)$, $1 \leq h \leq d+1$, son las funciones coordenadas baricéntricas con respecto a los a_j.

Por 5.27, tenemos que cada función coordenada baricéntrica es una función afín de $\mathbb{R}^d$ en $\mathbb{R}$ y se tiene

$$\forall x \in \mathbb{R}^d, \quad x = \sum_{j=1}^{d+1} \lambda_j(x) a_j.$$

Usando las coordenadas baricéntricas, el d-símplex K está caracterizado por

$$K = \{x \in \mathbb{R}^d;\ 0 \leq \lambda_j(x) \leq 1,\ 1 \leq j \leq d+1\}. \tag{5.28}$$

Para todo $k \geq 0$, denotaremos por $\mathbb{P}_k$ al espacio de polinomios de $\mathbb{R}^d$ en $\mathbb{R}$ de grado inferior o igual a k. Como la dimensión del espacio de polinomios de d variables de grado k es de dimensión $\begin{pmatrix} d+k-1 \\ k \end{pmatrix}$, la dimensión de $\mathbb{P}_k$ es

$$dim(\mathbb{P}_k) = \sum_{i=0}^{k} \begin{pmatrix} d+l-1 \\ l \end{pmatrix} = \begin{pmatrix} d+k \\ k \end{pmatrix} = \frac{(d+k)!}{d!\,k!}$$

Definimos

$$\begin{aligned} \Sigma_k &= \left\{x \in \mathbb{R}^d;\ \lambda_j(x) \in \{0, \tfrac{1}{k}, \ldots, \tfrac{k-1}{k}, 1\},\ 1 \leq j \leq d+1\right\}, \quad \forall k \geq 1, \\ \Sigma_0 &= \left\{x \in \mathbb{R}^d;\ \lambda_j(x) = \tfrac{1}{d+1},\ 1 \leq j \leq d+1\right\}. \end{aligned} \tag{5.29}$$

Es fácil de ver que se verifica

$$card(\Sigma_k) = \begin{pmatrix} (d+1)+k-1 \\ k \end{pmatrix} = \begin{pmatrix} d+k \\ k \end{pmatrix} = \frac{(d+k)!}{d!\,k!}, \quad \forall k \in \mathbb{N} \cup \{0\}.$$

Con estas definiciones, se tiene

Teorema 5.40. *Para todo entero $k \geq 0$, el conjunto Σ_k es $\mathbb{P}_k$-unisolvente.*

Demostración. Como acabamos de ver, se tiene

$$card(\Sigma_k) = \begin{pmatrix} d+k \\ k \end{pmatrix} = dim(\mathbb{P}_k). \tag{5.30}$$

Por la Observación 5.22, sabemos que nos basta describir las funciones de base de $(K, \mathbb{P}_k, \Sigma_k)$ para probar el resultado. Si $k = 0$, claramente la funcion de base es

$$p_0(x) = 1, \quad \forall x \in \mathcal{K}.$$

Supongamos entonces $k \geq 1$. Todo punto de Σ_k se escribe de la forma

$$a_\mu = \frac{1}{k} \sum_{j=1}^{d+1} \mu_j a_j, \quad \mu = (\mu_1, \mu_2, \ldots, \mu_d),$$

donde los coeficientes μ_j son enteros que cumplen

$$\mu_j \geq 0, \quad \sum_{j=1}^{d+1} \mu_j = k.$$

A un a_μ de Σ_k le vamos a asociar p_μ definido por

$$p_\mu(x) = \left(\prod_{j=1}^{d+1} (\mu_j!) \right)^{-1} \prod_{\substack{j=1 \\ \mu_j \geq 1}}^{d+1} \prod_{i=0}^{\mu_j - 1} (k\lambda_j(x) - i), \tag{5.31}$$

donde los λ_j son las funciones coordenadas baricéntricas respecto a los a_j, $1 \leq j \leq d+1$. Dado que los λ_j son afines, el p_μ definido por (5.31) es un polinomio de $\mathbb{P}_k$. Siendo las coordenadas baricéntircas de a_μ $\left(\frac{\mu_1}{k}, \ldots, \frac{\mu_{d+1}}{k}\right)$, tenemos que $p_\mu(a_\mu) = 1$. Sea ahora $a_v \in \Sigma_k$, $a_v \neq a_\mu$. Se tiene

$$a_v = \frac{1}{k} \sum_{j=1}^{d+1} v_j a_j, \quad v = (v_1, v_2, \ldots, v_d),$$

$$p_\mu(a_v) = \left(\prod_{j=1}^{d+1} (\mu_j!) \right)^{-1} \prod_{\substack{j=1 \\ \mu_j \geq 1}}^{d+1} \prod_{i=0}^{\mu_j - 1} (v_j - i)$$

Como $a_\mu \neq a_v$, existe un índice j_0, $1 \leq j_0 \leq d$, tal que $v_{j_0} < \mu_{j_0}$. Como son enteros, se tiene

$$\mu_{j_0} \geq v_{j_0} + 1,$$

lo que implica $\mu_{j_0} \geq 1$; $v_{j_0} \geq 0$; $v_{j_0} - (\mu_{j_0} - 1) \leq 0$. de donde concluímos

$$p_\mu(a_v) = 0,$$

lo que demuestra que las funciones p_μ definidas en (5.31) son las funciones de base de $(K, \mathbb{P}_k, \Sigma_k)$ concluyendo así la prueba. ∎

Definición 5.41. *Para todo símplice K de $\mathbb{R}^d$ y para todo entero $k \geq 0$, el elemento finito $(K, \mathbb{P}_k, \Sigma_k)$, con Σ_k dado por (5.29), se llama d-símplex de tipo (k).*

Teorema 5.42. *Para todo entero $k \geq 0$, dos elementos finitos d-simpliciales de tipo (k) son afín-equivalentes.*

Demostración. Sean $(\hat{K}, \hat{\mathbb{P}}_k, \hat{\Sigma}_k)$ y $(K, \mathbb{P}_k, \Sigma_k)$ dos elemento finitos simpliciales de tipo (k). Denotaremos:

$$\begin{aligned} \hat{K} &= d - \text{simplex de vértices } \hat{a}_j, \quad 1 \leq j \leq d+1, \\ K &= d - \text{simplex de vértices } a_j, \quad 1 \leq j \leq d+1, \end{aligned}$$

Tomamos la aplicación $F : \mathbb{R}^d \to \mathbb{R}^d$ que a cada $\hat{x} \in \mathbb{R}^d$ de coordenadas baricéntricas $\hat{\lambda}_1, \ldots, \hat{\lambda}_{d+1}$ respecto a $\hat{a}_j$, $1 \leq j \leq d+1$, le asocia el punto $x = F(\hat{x})$ de coordenadas baricéntricas $\lambda_1, \ldots, \lambda_{d+1}$ respecto a a_j, $1 \leq j \leq d+1$, tal que

$$\lambda_j = \hat{\lambda}_j, \; 1 \leq j \leq d+1. \tag{5.32}$$

Sean $(\hat{a}_{ij})_{i=1}^d$ las coordenadas baricéntricas del vértice $\hat{a}_j$ de $\hat{K}$ y $\hat{A}$ la matriz inversible de orden $d+1$

$$\hat{A} = \begin{pmatrix} \hat{a}_{11} & \hat{a}_{12} & \ldots & \hat{a}_{1,d+1} \\ \vdots & \vdots & & \vdots \\ \hat{a}_{d1} & \hat{a}_{d2} & \ldots & \hat{a}_{d,d+1} \\ 1 & 1 & \ldots & 1 \end{pmatrix}.$$

Entonces $F : \mathbb{R} \to \mathbb{R}^d$ es la aplicación que, a un punto $\hat{x}$ de coordenadas cartesianas $(\hat{x}_i)_{1 \leq i \leq d}$, le asocia el punto $x = F(\hat{x})$ de coordenadas cartesianas $(x_i)_{1 \leq i \leq d}$ definidas por

$$\begin{pmatrix} x_1 \\ \vdots \\ x_d \\ 1 \end{pmatrix} = A \begin{pmatrix} \lambda_1 \\ \vdots \\ \vdots \\ \lambda_{d+1} \end{pmatrix} = A \begin{pmatrix} \hat{\lambda}_1 \\ \vdots \\ \vdots \\ \hat{\lambda}_{d+1} \end{pmatrix} = A\hat{A}^{-1} \begin{pmatrix} \hat{x}_1 \\ \vdots \\ \hat{x}_d \\ 1 \end{pmatrix}.$$

Se tiene, por tanto,

$$x = B\hat{x} + b, \tag{5.33}$$

donde B es una matriz $d \times d$ invertible y $b \in \mathbb{R}^d$. Luego F es una aplicación afín inversible. Luego, se tiene $K = F(\hat{K})$, $\Sigma_k = F(\hat{\Sigma}_k)$ y que $\mathbb{P}_k$ es globalmente invariante por F y, en virtud de la Definición 5.35, los elementos finitos $(\hat{K}, \hat{P}, \hat{\Sigma})$ y $(K, \mathbb{P}_k, \Sigma_k)$ son afínmente equivalentes, probando así el resultado. ∎

Terminaremos este apartado viendo los símplices de tipo (k) más comunes:

Para $d = 2$ y a_1, a_2, a_3 dados, K es el triángulo de vértices a_1, a_2, a_3.

- Para $k = 0$, el triángulo de tipo (0) corresponde a tomar $P = \mathbb{P}_0$ y $\Sigma = \Sigma_0$ como el baricentro del triángulo.
- Para $k = 1$, el triángulo de tipo (1) corresponde a tomar $P = \mathbb{P}_1$ y $\Sigma = \Sigma_1$ como los vértices del triángulo.

- Para $k = 2$, el triángulo de tipo (2) corresponde a tomar $P = \mathbb{P}_2$ y $\Sigma = \Sigma_2$ como los vértices y los puntos medios de los lados del triángulo.

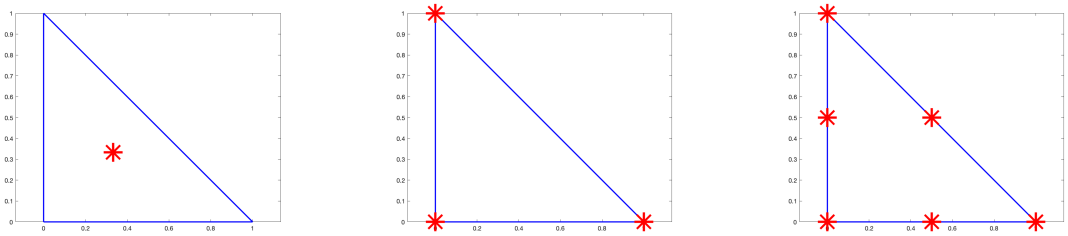

Figura 5.1: Representación en matlab de los triángulos de tipo (0), (1) y (2)

Para $d = 3$ y a_1, a_2, a_3, a_4 dados, K es el tetraedro de vértices a_1, a_2, a_3, a_4.

- Para $k = 0$, el tetraedro de tipo (0) corresponde a tomar $P = \mathbb{P}_0$ y $\Sigma = \Sigma_0$ como el baricentro del tetraedro.
- Para $k = 1$, el tetraedro de tipo (1) corresponde a tomar $P = \mathbb{P}_1$ y $\Sigma = \Sigma_1$ como los vértices del tetraedro.
- Para $k = 2$, el tetraedro de tipo (2) corresponde a tomar $P = \mathbb{P}_2$ y $\Sigma = \Sigma_2$ como los vértices y los puntos medios de los lados del tetraedro.

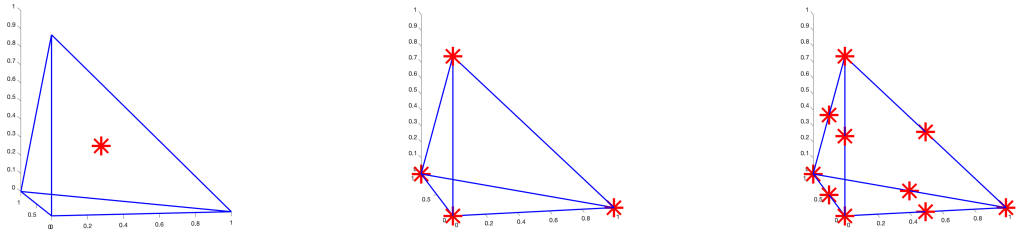

Figura 5.2: Representación en matlab de los tetraedros de tipo (0), (1) y (2)

5.4.3. Elementos finitos paralelotopos

Otro tipo de elementos finitos muy similares a los vistos en la sección anterior son los elementos finitos paralelotopos. En este caso, el elemento de referencia $\hat{K} = [0,1]^d$. Para k entero no negativo, tomamos $\mathbb{Q}_k$ el espacio de polinomios de grado igual o inferior a k en cada variable y $\hat{\Sigma}_k$ el conjunto

$$\hat{\Sigma}_k = \left\{ \hat{x} = (\hat{x}_i)_{1 \le i \le d};\ \hat{x}_i \in \left\{ 0, \frac{1}{k}, \ldots, \frac{k-1}{k}, 1 \right\},\ 1 \le i \le d \right\}, \tag{5.34}$$

que es $\mathbb{Q}_k$-unisolvente.

De forma análoga al apartado anterior, vamos a mostrar los $\mathbb{Q}_k$ de tipo (0), (1) y (2) en dimensiones 2 y 3.

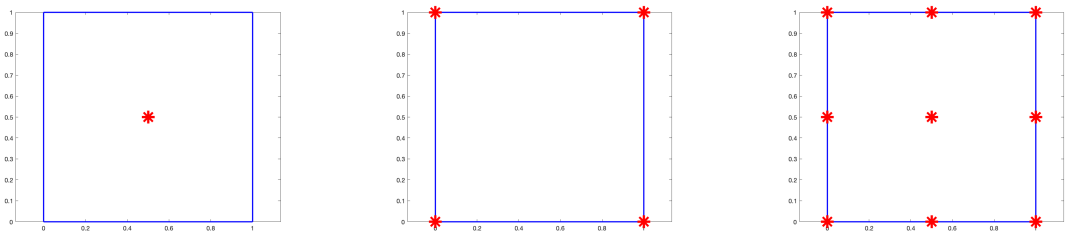

Figura 5.3: Representación en matlab de los cuadrados de tipo (0), (1) y (2)

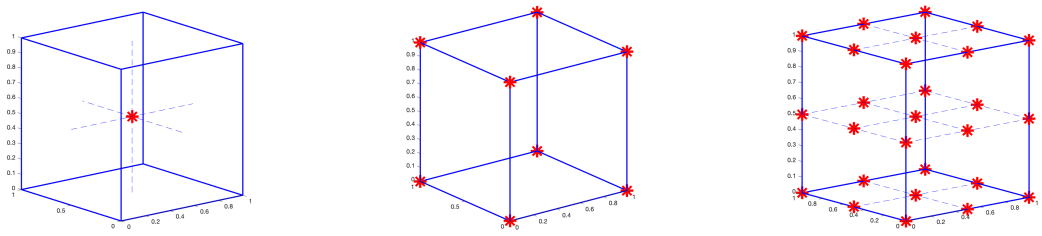

Figura 5.4: Representación en matlab de los cubos de tipo (0), (1) y (2)

5.5. Estimaciones de error

En la sección 5.4.1, para un elemento finito de Lagrange (K, P, Σ) y una función v definida en K, hemos Πv por (5.21). En esta sección, para una función $v \in H^{k+1}(\overset{\circ}{K})$, con k un entero positivo, vamos a estudiar el error de interpolación $v - \Pi v$ en la norma de $H^m(\overset{\circ}{K})$, con $0 \leq m \leq k+1$. Observemos que, para que Πv esté bien definida, tenemos que poder evaluar v en los puntos a_i, por lo que necesitaremos que K sea suficientemente rico, de modo que $H^{k+1}(\overset{\circ}{K})$ esté contenido en $\mathcal{C}^0(K)$. Por simplificar, denotaremos por $H^m(K)$ el espacio de Sobolev $H^m(\overset{\circ}{K})$.

Comenzamos con la siguiente definición:

Definición 5.43. *Para un subespacio $E \subset H^m(K)$, definimos el espacio cociente $H^m(K)/E$ como el espacio de clases de equivalencia $\dot{v}$ de las funciones de $H^m(K)$ donde la relación de equivalencia $\Re$ viene dada por:*

$$v_1 \Re v_2 \iff v_1 - v_2 \in E.$$

Este espacio es normado con la norma

$$\|\dot{v}\|_{H^m(K)/E} = \inf_{v \in \dot{v}} \|v\|_{m,K}. \tag{5.35}$$

Cuando E es un subespacio cerrado de $H^m(K)$, se tiene que $H^m(K)/E$ es un espacio de Hilbert. En particular, como el espacio $\mathbb{P}_k$ es un subespacio de dimensión finita

de $H^{k+1}(K)$, y por tanto cerrado, se tiene que el espacio cociente $H^{k+1}(K)/\mathbb{P}_k$ es un espacio de Hilbert.

Vamos a dotar a ese espacio de una norma más manejable que la norma (5.35); para ello, vamos a suponer que la inyección canónica de $H^1(K)$ en $L^2(K)$ es compacta (hipótesis que se verifica tanto para los $\mathbb{P}_k$ como para los $\mathbb{Q}_k$. Dado que K es un acotado de $\mathbb{R}^d$, esta propiedad se verifica cuando la frontera de K sea de $\mathcal{C}^1$ por partes en virtud del Teorema 5.12.

Teorema 5.44. *Sea K un compacto conexo de frontera $\mathcal{C}^1$ por partes. Entonces, para todo k entero no nulo, la aplicación*

$$\dot{v} \mapsto |\dot{v}|_{k+1,K} = |v|_{k+1,K}, \tag{5.36}$$

donde v es un representante cualquiera de $\dot{v}$ en $H^{k+1}(K)/\mathbb{P}_k$, es una norma sobre $H^{k+1}(K)/\mathbb{P}_k$ equivalente a la norma cociente.

La demostración de este resultado utiliza el argumento clásico de compacidad-unicidad teniendo en cuenta la compacidad de la inmersión de $H^{k+1}(K)$ en $H^k(K)$ para todo k entero no negativo (ver [5, págs 96-98])

Del teorema anterior deducimos un resultado general de error entre espacios de Sobolev y espacios de polinomios.

Teorema 5.45. *Sea K un compacto conexo de frontera $\mathcal{C}^1$ por partes y sea Π un operador lineal continuo de $H^{k+1}(K)$ en $H^m(K)$, $0 \leq m \leq k+1$, tal que*

$$\Pi p = p, \quad \forall p \in \mathbb{P}_k \tag{5.37}$$

Entonces, existe una constante $c = c(K, \Pi)$ tal que

$$\|v - \Pi v\|_{m,K} \leq c|v|_{k+1,K}, \quad \forall v \in H^{k+1}(K). \tag{5.38}$$

Demostración. Sea v una función de $H^{k+1}(K)$. Por (5.37), podemos escribir para todo polinomio p de $\mathbb{P}_k$

$$v - \Pi v = v + p - \Pi(v + p),$$

de donde

$$\|v - \Pi v\|_{m,K} \leq c^\star \|v + p\|_{k+1,K}$$

con

$$c^\star = \|I - \Pi\|_{\mathcal{L}(H^{k+1}(K);H^m(K))}. \tag{5.39}$$

Entonces

$$\|v - \Pi v\|_{m,K} \leq c^\star \inf_{p \in \mathbb{P}_k} \|v + p\|_{k+1,K}, \quad \forall v \in H^{k+1}(K),$$

de donde, por la equivalencia de las normas dada por el Teorema 5.44, en particular se tiene

$$\|v - \Pi v\|_{m,K} \leq c^\star C |v|_{k+1,K}, \quad \forall v \in H^{k+1}(K),$$

lo que prueba (5.38) con

$$c = c^\star C. \tag{5.40}$$

Para dar la dependencia de c respecto a las características geométricas de K, vamos a considerar $\hat{K}$ compacto de $\mathbb{R}^d$, conexo y de frontera $\mathcal{C}^1$ por partes, que nos va a servir como dominio de referencia.

Supongamos que existe una transformación afín-inversible

$$x = F(\hat{x}) = B\hat{x} + b, \tag{5.41}$$

donde B es una matriz de orden d inversible y b un vector de $\mathbb{R}^d$, tal que

$$K = F(\hat{K}). \tag{5.42}$$

Vamos a adoptar los convenios de notación (5.39)-(5.40): a toda función v definida sobre K se le asocia biunívocamente la función $\hat{v}$ definida sobre $\hat{K}$ por

$$\hat{v}(\hat{x}) = v(x), \quad \forall \hat{x} \in \hat{K}.$$

Nos será de utilidad introducir las características geométricas siguientes:

$$h_K = \text{máx}\left\{|x-y| : \ x, y \in K\right\}, \tag{5.43}$$

$$\rho_K = \text{máx}\left\{r > 0 : \ \exists x \in \mathbb{R}^d, \ B(x,r) \subset K\right\}, \tag{5.44}$$

que en el caso particular de $\hat{K}$ se denotarán por $\hat{h}$ y $\hat{\rho}$, respectivamente.

Entonces podemos evaluar las normas espectrales $\|B\|$ y $\|B^{-1}\|$ en función de las características geométricas de K y $\hat{K}$. Recordemos que la norma espectral de una matriz B se define como

$$\|B\| = \sup_{\xi \in \mathbb{R}^d, \xi \neq 0} \frac{|B\xi|}{|\xi|},$$

donde $|\xi|$ es la norma euclídea de $\xi \in \mathbb{R}^d$. El siguiente lema acota las normas espectrales de B y B^{-1}:

Lema 5.46. Se tienen las acotaciones

$$\|B\| \leq \frac{1}{\hat{\rho}} h_k, \quad \|B^{-1}\| \leq \frac{1}{\rho_k} \hat{h}. \tag{5.45}$$

La demostración de este lema puede encontrarse en el capítulo 4 de [5]. El resultado principal de este epígrafe viene dado por

Teorema 5.47. *Sea $\hat{K}$ un compacto conexo de frontera $\mathcal{C}^1$ por partes y sea $\hat{\Pi}$ un operador lineal continuo de $H^{k+1}(\hat{K})$ en $H^m(\hat{K})$, $0 \leq m \leq k+1$, tal que*

$$\hat{\Pi}\hat{p} = \hat{p}, \quad \forall \hat{p} \in \mathbb{P}_k. \tag{5.46}$$

Si K es un conjunto de $\mathbb{R}^d$ tal que existe una transformación afín inversible $F : \mathbb{R}^d \to \mathbb{R}^d$ para la cual K es la imagen de F por $\hat{K}$ y si el operador Π está definido por

$$\Pi\hat{v} = \hat{\Pi}\hat{v}, \quad \forall v \in H^{k+1}(K), \tag{5.47}$$

entonces existe una constante C independiente de F tal que

$$|v - \Pi v|_{m,K} \leq C \frac{h_K^{k+1}}{\rho_K^m}, \quad \forall v \in H^{k+1}(K). \tag{5.48}$$

Demostración. Si $(e_1, ..., e_d)$ es la base canónica de $\mathbb{R}^d$ y si $\alpha = (\alpha_1, \ldots, \alpha_d) \in \mathbb{N}^d$, tenemos

$$\partial^\alpha v(x) \cdot (\underbrace{e_1, \ldots, e_1}_{\alpha_1 \text{ veces}}, \ldots, \underbrace{e_d, \ldots, e_d}_{\alpha_d \text{ veces}}), \quad \text{con } |\alpha| = \alpha_1 + \ldots + \alpha_d.$$

Luego existen unas constantes $\gamma_i = \gamma_i(l, d) > 0$, $i = 1, 2$, tales que

$$\gamma_1 |v|_{l,K} \leq \left(\int_K \|D^l v(x)\|^2 \, dx \right)^{\frac{1}{2}} \leq \gamma_2 |v|_{l,K}, \quad \forall v \in \mathcal{D}(K). \tag{5.49}$$

Sea ahora v una función de $\mathcal{D}(K)$ y $\hat{v} = v \circ F$ con F una aplicación afín invertible en las condiciones de (5.41). La derivada i-ésima de $\hat{v}$ verifica:

$$D^l \hat{v}(\hat{x}) \cdot (\xi_1, \ldots, \xi_l) = D^l v(x) \cdot (B\xi_1, \ldots, B\xi_l), \forall \xi_1, \ldots, \xi_l \in \mathbb{R}^d,$$

de donde

$$\|D^l \hat{v}(\hat{x})\| \leq \|B\|^l \|D^l v(x)\|$$

por lo que

$$\int_{\hat{K}} \|D^l \hat{v}(\hat{x})\|^2 \, d\hat{x} \leq \|B\|^{2l} \int_{\hat{K}} \|D^l v(F(\hat{x}))\|^2 \, d\hat{x}.$$

Usando la fórmula de cambio de variables para escribir la segunda integral en función de x, obtenemos

$$\int_{\hat{K}} \|D^l \hat{v}(\hat{x})\|^2 \, d\hat{x} \leq \|B\|^{2l} |B|^{-1} \int_K \|D^l v(x)\|^2 \, dx.$$

Usando (5.49), deducimos

$$|\hat{v}|_{l,\hat{K}} \leq \gamma \|B\|^l |B|^{-\frac{1}{2}} |v|_{l,K}, \quad \forall v \in \mathcal{D}(K),$$

tomando

$$\gamma = \frac{\gamma_2}{\gamma_1} > 0.$$

Cuando $\mathcal{D}(K)$ es denso en $H^l(K)$, tenemos

$$|\hat{v}|_{l,\hat{K}} \leq \gamma \|B\|^l |B|^{-\frac{1}{2}} |v|_{l,K}, \quad \forall v \in H^l(K), \tag{5.50}$$

con una constante γ que solo depende de l y d.

Solo hemos probado que $\mathcal{D}(K)$ es denso en $H^1(K)$. Podemos obtener (5.50) por recurrencia sobre l ya que tenemos que

$$|v|_{l,K} = \left(\sum_{i=1}^d \left| \frac{\partial v}{\partial x_i} \right|_{l-1,K}^2 \right)^{\frac{1}{2}}.$$

Análogamente, cambiando los roles de K y $\hat{K}$, obtenemos

$$|v|_{l,K} \leq \gamma \|B\|^l |B|^{-\frac{1}{2}} |\hat{v}|_{l,\hat{K}}, \quad \forall \hat{v} \in H^l(\hat{K}). \tag{5.51}$$

Sea ahora v una función de $H^{k+1}(K)$. De (5.51), deducimos

$$|v - \Pi v|_{m,K} \leq \gamma(m,d)\|B^{-1}\|^m |B|^{\frac{1}{2}} |\hat{v} - \hat{\Pi}v|_{m,\hat{K}}.$$

En virtud de las hipótesis (5.45) y (5.46), podemos aplicar el Teorema 5.45 en $\hat{K}$, lo que nos da

$$|\hat{v} - \hat{\Pi}v|_{m,\hat{K}} = |\hat{v} - \hat{\Pi}\hat{v}|_{m,\hat{K}} \leq c(\hat{K},\hat{\Pi})|\hat{v}|_{k+1,\hat{K}},$$

donde $c(\hat{K},\hat{\Pi})$ es una constante que solo depende de $\hat{K}$ y $\hat{\Pi}$. Por (5.50)

$$|\hat{v}|_{k+1,\hat{K}} \leq \gamma(k+1,d)\|B\|^{k+1}|B|^{-\frac{1}{2}}|v|_{k+1,K}.$$

Luego

$$|v - \Pi v|_{m,K} \leq \gamma(m,d)\gamma(k+1,d)c(\hat{K},\hat{\Pi})\|B\|^{k+1}\|B^{-1}\|^m |v|_{k+1,K}.$$

Finalmente, basta con aplicar el Lema 5.46 para obtener la cota buscada (5.48) con

$$C = \gamma(m,d)\gamma(k+1,d)\hat{h}^m \left(\frac{1}{\hat{\rho}}\right)^{k+1} c(\hat{K},\hat{\Pi}). \tag{5.52}$$

Observación 5.48. La constante C depende de $\hat{K}$ (y, por tanto, de d) y de $\hat{\Pi}$ (luego también de k y m). La novedad esencial respecto al Teorema 5.45 reside en la obtención de una cota de $|v - \Pi v|_{m,K}$ donde la constante C no depende de F y, por consiguiente, de las características geométricas del dominio K. De este modo, se tiene para todo m, $0 \leq m \leq k+1$:

$$\|v - \Pi v\|_{m,K} \leq C\frac{h_K^{k+1}}{\rho_K^m}|v|_{k+1,K}, \quad \forall v \in H^{k+1}(K)$$

con C independiente de K.

En el caso $d \leq 3$, $H^2(K)$ se inyecta de forma continua en $\mathcal{C}^0(K)$. Teniendo en cuenta el Teorema 5.47, se puede probar el siguiente resultado:

Teorema 5.49. *Sean (K,P,Σ) y $(\hat{K},\hat{P},\hat{\Sigma})$ dos elementos finitos afín-equivalentes. Supongamos que K es un compacto y conexo de $\mathbb{R}^d$, $d \leq 3$, de frontera $\mathcal{C}^1$ por trozos. Además, supongamos que existe un entero $k \geq 1$ tal que*

$$\mathbb{P}_k \subset P \subset H^{k+1}(K). \tag{5.53}$$

Entonces existe una constante C que solamente depende del elemento finito $(\hat{K},\hat{P},\hat{\Sigma})$ tal que para todo entero m, $0 \leq m \leq k+1$, se tiene

$$|v - \Pi v|_{m,K} \leq C\frac{h_K^{k+1}}{\rho_K^m}|v|_{k+1,K}, \quad \forall v \in H^{k+1}(K), \tag{5.54}$$

donde Π es el operador de P-interpolación sobre Σ.

Corolario 5.50. Sea (K, P, Σ) un d-símplex de tipo (k). Supongamos $d \leq 3$ y $k \geq 1$. Entonces existe una constante C que solo depende de d y k tal que para todo entero m, $0 \leq m \leq k+1$, tenemos

$$|v - \Pi v|_{m,K} \leq C \frac{h_K^{k+1}}{\rho_K^m} |v|_{k+1,K}, \quad \forall v \in H^{k+1}(K).$$

Corolario 5.51. Sea (K, P, Σ) un d-paralelotopo de tipo (k). Supongamos $d \leq 3$ y $k \geq 1$. Entonces existe una constante C que solo depende de d y k tal que para todo entero m, $0 \leq m \leq k+1$, tenemos

$$|v - \Pi v|_{m,K} \leq C \frac{h_K^{k+1}}{\rho_K^m} |v|_{k+1,K}, \quad \forall v \in H^{k+1}(K).$$

5.6. Análisis del método de los elementos finitos

Utilizando los elementos finitos de Lagrange construidos en 5.4, vamos a desarrollar una teoría general de aproximación de problemas elípticos de segundo orden. Vamos a estudiar métodos de aproximación de orden k en el caso de un abierto poliédrico, es decir, unión finita de d-símplices. En la práctica, abiertos más generales suficientemente regulares, se aproximan por abiertos poliédricos.

Dado $\Omega \subset \mathbb{R}^d$ un abierto poliédrico, consideramos una descomposición de $\overline{\Omega}$ del tipo

$$\overline{\Omega} = \bigcup_{K \in \mathcal{T}_h} K, \tag{5.55}$$

tal que

1. cada elemento K de $\mathcal{T}_h$ es un poliedro de $\mathbb{R}^d$ de interior no vacío;
2. los interiores de dos poliedros de $\mathcal{T}_h$ distintos son disjuntos;
3. toda cara de un poliedro $K_1 \in \mathcal{T}_h$ es o bien una cara de otro poliedro $K_2 \in \mathcal{T}_h$, en cuyo caso se dice que K_1 y K_2 son adyacentes, o bien parte de la frontera Γ de Ω.

Definición 5.52. *Toda descomposición de $\overline{\Omega}$ cumpliendo las propiedades anteriores, se llama triangulación de $\overline{\Omega}$.*

Denotamos por $\mathcal{T}_h$ una triangulación de $\overline{\Omega}$ de diámetro h, donde

$$h = \max_{K \in \mathcal{T}_h} h_K, \tag{5.56}$$

con h_K el diámetro del poliedro K.

Supongamos ahora que a cada poliedro K de $\mathcal{T}_h$ se le asocia un elemento finito de Lagrange (K, P_k, Σ_k) tal que

$$P_k \subset H^1(K), \tag{5.57}$$

y definidmos los espacios de dimensión dinita

$$X_h = \left\{ v \in \mathcal{C}^0(\overline{\Omega});\, \forall K \in \mathcal{T}_h,\, v|_h \in P_k \right\} \tag{5.58}$$

y

$$X_{oh} = \left\{ v \in X_h;\, v|_\Gamma = 0 \right\}. \tag{5.59}$$

Se tiene, por sus definiciones, $X_h \subset H^1(\Omega)$ y $X_{oh} \subset H_0^1(\Omega)$. La teoría de aproximación variacional nos conduce entonces a buscar u_h solución en V_h del problema $(\mathcal{P}_h)$:

$$\mathfrak{a}(u_h, v_h) = L(v_h), \quad \forall v_h \in V_h, \tag{5.60}$$

con

$$\begin{array}{l} V_h = X_h \quad \text{si} \quad V = H^1(\Omega), \\ V_h = X_{oh} \quad \text{si} \quad V = H_0^1(\Omega). \end{array}$$

Vamos a introducir el operador Π_h que a toda función v continua sobre $\overline{\Omega}$ le asocia la función $\Pi_h v$ de $L^2(\Omega)$ definida por

$$\Pi_h v(x) = \Pi_K v(x), \quad \forall K \in \mathcal{T}_h,\, \forall x \in \overset{\circ}{K}, \tag{5.61}$$

donde Π_K es el operador de P_K-interpolación sobre Σ_K.

Consideramos la siguiente hipótesis de compatibilidad. Supongamos que para todo par K_1, K_2 de poliedros de $\mathcal{T}_h$ adyacentes, de cara común $K' = K_1 \cap K_2$, tenemos

$$\begin{array}{c} P_{K_1}|_{K'} = P_{K_2}|_{K'}, \\ \Sigma_{K_1} \cap K' = \Sigma_{K_2} \cap K'. \end{array} \tag{5.62}$$

Definición 5.53. *Sea K un poliedro de $\mathbb{R}^d$; un elemento finito (K, P, Σ) se dice que es de clase $\mathcal{C}^0$ si se satisfacen las condiciones siguientes:*

1. $$P \subset \mathcal{C}^0(K), \tag{5.63}$$

2. *para toda cara K' de K, el conjunto $\Sigma' = \Sigma \cap K'$ es P'-unisolvente, donde $P' = \{p|_{K'} : p \in P\}$*

Gracias a las condiciones de compatibilidad, se puede probar (Capítulo 5 [5]) que todo elemento finito afín-equivalente a un elemento finito de clase $\mathcal{C}^0$ es también de $\mathcal{C}^0$. Concretamente, se tiene

Teorema 5.54. *Sea $\mathcal{T}_h$ una triangulación de $\overline{\Omega}$ y sea $(K, P_K, \Sigma_K)_{K\in\mathcal{T}_h}$ una familia de elementos finitos asociada. Supongamos que las condiciones de compatibilidad (5.62) se satisfacen y que para todo $K \in \mathcal{T}_h$, (K, P_K, Σ_K) es un elemento finito de clase $\mathcal{C}^0$ y P_K un subespacio de $H^1(K)$.*

Entonces, el operador de interpolación Π_h, definido en (5.61) de $\mathcal{C}^0(\overline{\Omega})$ en $L^2(\Omega)$ toma valores en $\mathcal{C}^0(\overline{\Omega})$; de forma más precisa, tenemos

$$\begin{aligned} X_h &= \left\{\Pi_h v;\ v \in \mathcal{C}^0(\overline{\Omega})\right\}, \\ X_{oh} &= \left\{\Pi_h v;\ v \in \mathcal{C}^0(\overline{\Omega});\ v|_\Gamma = 0\right\} \end{aligned} \tag{5.64}$$

donde X_h y X_{oh} son los subespacios de $\mathcal{C}^0(\overline{\Omega})$ introducidos en (5.58) y (5.59).

Vamos a introducir el conjunto de grados de libertad de los elementos finitos

$$\Sigma_h = \bigcup_{K\in\mathcal{T}_h} \Sigma_K \tag{5.65}$$

Recordemos que Σ_K se identifica con su aplicación y, por tanto, con unos a_i. Numeremos los elementos de Σ_h de modo que queden al final los puntos pertenecientes a la frontera Γ de forma que podamos escribir

$$\Sigma_h = \{a_i\}_{1\le i\le I} \text{ con } \operatorname{card}(\Sigma_h) = I, \tag{5.66}$$

y

$$\Sigma_{oh} := \Sigma_h \cap \Omega = \{a_i\}_{1\le i\le I} \text{ con } \operatorname{card}(\Sigma_{oh}) = I_0 < I. \tag{5.67}$$

Para todo entero i, $1 \le i \le I$, denotemos por φ_i la función de X_h tal que

$$\varphi_i(a_j) = \delta_{ij}, \quad 1 \le j \le I. \tag{5.68}$$

Si v es una función continua sobre $\overline{\Omega}$, claramente

$$\Pi_h v = \sum_{i=1}^{I} v(a_i)\varphi_i, \tag{5.69}$$

de donde deducimos el siguiente corolario:

Corolario 5.55. Sobre las hipótesis del Teorema 5.54, el conjunto de funciones φ_i, $1 \le i \le I$, definidas en (5.68), forman una base de X_h. Asimismo, las funciones φ_i, $1 \le i \le I_0$ forman una base de X_{oh}.

Definición 5.56. *Los escalares $v(a_i)$ que aparecen al escribir una función v de X_h (o X_{oh})en combinación lineal de los elementos de la base de los φ_i se denominan grados de libertad de v.*

En la práctica, la introducción de los φ_i es útil ya que tienen un soporte pequeño, de modo que habrá muchos términos nulos en la matriz de Galerkin correspondiente, lo que será fundamental desde el punto de vista computacional. De hecho, el soporte de φ_i es el conjunto de los $K \in \mathcal{T}_h$ a los que pertenece el nodo a_i.

Antes de pasar a estudiar la convergencia de la solución u_h de (5.60), veamos algunos ejemplos donde se verifican las hipótesis del Teorema 5.54:

Ejemplo 5.57. Sea $\mathcal{T}_h$ una triangulación de $\overline{\Omega}$ construida usando d-símplices. Dado un entero positivo k, asociamos a todo K de $\mathcal{T}_h$ el d-símplice de tipo (k) construido sobre K.

Ejemplo 5.58. Sea $\mathcal{T}_h$ una triangulación de $\overline{\Omega}$ construida usando d-paralelotopos. Dado un entero positivo k, asociamos a todo K de $\mathcal{T}_h$ el d-paralelotopo de tipo (k) construido sobre K.

Observación 5.59. Claramente, podemos generalizar estos ejemplos mediante triangulaciones donde cada elemento finito sea o bien un d-símplice o bien un d-paralelotopo de tipo (k).

Hemos visto que el cálculo efectivo de la aproximación u_h es realizable numéricamente. Nos falta por ver que la aproximación obtenida converge a la solución u del problema $(\mathcal{P})$. Para ello, introducimos la siguiente definición:

Definición 5.60. *Diremos que $(\mathcal{T}_h)_{h\geq 0}$ es una familia regular de triangulaciones de $\overline{\Omega}$ si se satisfacen las siguientes condiciones:*

1. *Todos los elementos finitos $(K, \mathbb{P}_K, \Sigma_K)$ de todas las triangulaciones son afín-equivalentes a un mismo elemento finito de referencia $(\hat{K}, \hat{P}, \hat{\Sigma})$ de clase $\mathcal{C}^0$.*
2. *Para toda pareja $(\hat{K}_1', \hat{K}_2')$ de caras de $\hat{K}$ y para toda aplicación $\hat{F} : \mathbb{R}^d \to \mathbb{R}^d$ afínmente invertible tal que $\hat{K}_2' = \hat{F}(\hat{K}_1')$, tenemos*

$$\hat{\Sigma} \cap \hat{K}_2' = \hat{F}(\hat{\Sigma} \cap \hat{K}_1')$$
$$\{\hat{p}|_{K_2'};\ \hat{p} \in \hat{P}\} = \{p \circ \hat{F}|_{K_1'};\ p \in \hat{P}\}.$$

3. *Se tiene*

$$\lim_{h\to 0} T_h = \overline{\Omega}$$

4. *Existe una constante $\sigma \geq 1$ tal que*

$$\frac{h_K}{\rho_K} \leq \sigma \quad \forall h,\ \forall K \in \mathcal{T}_h. \tag{5.70}$$

Se tiene el siguiente teorema:

Teorema 5.61. *Sea Ω un abierto poliédrico de $\mathbb{R}^d$, $d \leq 3$. Sea $(\mathcal{T}_h)$ una familia regular de triangulaciones de $\overline{\Omega}$ asociada a un elemento finito de referencia $(\hat{K}, \hat{P}, \hat{\Sigma})$ de clase $\mathcal{C}^0$. Supongamos que existe un entero positivo k verificando*

$$\mathbb{P}_k \subset \hat{P} \subset H^1(\hat{K}). \tag{5.71}$$

Entonces el método de elementos finitos es convergente, es decir, la solución u_h del problema $(\mathcal{P}_h)$ converge a la solución u de $(\mathcal{P})$ en $H^1(\Omega)$:

$$\lim_{h\to 0} \|u - u_h\|_{1,\Omega} = 0 \tag{5.72}$$

Este método es de orden (al menos) k, es decir, existe una constante C independiente de h tal que, si la solución u pertenece a $H^{k+1}(\Omega)$, tenemos

$$\|u - u_h\|_{1,\Omega} \leq Ch^k |u|_{k+1,\Omega}. \tag{5.73}$$

Demostración. Comencemos por estudiar el caso donde u pertenece a $H^{k+1}(\Omega)$. u se puede probar continua aplicando de forma reiterada el Teorema 5.12 a las derivadas de u. Por tanto, podemos construir $\Pi_h u$. Por el Teorema 5.54, $\Pi_h u$ pertenecen al subespacio V_h de V ($V_h = X_h$ si $V = H^1(\Omega)$ y $V_h = X_{oh}$ si $V = H_0^1(\Omega)$). El Teorema 5.27 nos da

$$\|u - u_h\|_{1,\Omega} \leq C_1 \|u - \Pi_h u\|_{1,\Omega}$$

con $C_1 = \frac{M}{\alpha}$. Como la restricción de $\Pi_h u$ a todo $K \in \mathcal{T}_h$ es por definición la $\mathbb{P}_k$-interpolada de $\Pi_K u$ sw u sobre Σ_K, tenemos

$$\|u - \Pi_h u\|_{1,\Omega} = \left(\sum_{K \in \mathcal{T}_h} \|u - \Pi_K u\|_{1,K}^2 \right)^{1/2}.$$

Sin embargo, de acuerdo con el Teorema 5.49, existen dos constantes C_2 y C_3 que solo dependen del elemento finito de referencia $(\hat{K}, \hat{P}, \hat{\Sigma})$ tales que

$$\|u - \Pi_K u\|_{1,K} \leq C_2 \frac{h_K^{k+1}}{\rho_K} |u|_{k+1,K}$$

y

$$\|u - \Pi_K u\|_{0,K} \leq C_3 h_K^{k+1} |u|_{k+1,K}.$$

Siendo regular $\mathcal{T}_h$, deducimos

$$\|u - \Pi_K u\|_{0,K} \leq C_4 h^k |u|_{k+1,K},$$

con $C_4 = \sigma \left\{ C_2^2 + (C_3 \operatorname{diam}(\overline{\Omega}))^2 \right\}^{1/2}$, donde σ es la constante que aparece en (5.70). Así, una estimación del error de interpolación en $\overline{\Omega}$ es

$$\|u - \Pi_h u\|_{1,\Omega} \leq C_4 h^k \left(\sum_{K \in \mathcal{T}_h} |u|_{k+1,K}^2 \right)^{1/2} = C_4 h^k |u|_{k+1,\Omega}. \tag{5.74}$$

La cota (5.73) se obtiene así tomando $C = C_1 C_4$. Para probar la convergencia del método, usaremos la Observación 5.29

$$\mathcal{V} = \mathcal{D}(\overline{\Omega}) \text{ si } V = H^1(\Omega), \ \mathcal{V} = \mathcal{D}(\Omega) \text{ si } V = H_0^1(\Omega),$$

y

$$r_h = \Pi_h.$$

Para toda función v de $\mathcal{V}$, se tiene (5.74), luego

$$\lim_{h \to 0} \|v - \Pi_h v\|_{1,\Omega} = 0,$$

lo que prueba el resultado ya que $\mathcal{V}$ es denso en V. ∎

A continuación, enunciamos dos corolarios fundamentales del Teorema 5.61.

Corolario 5.62. Supongamos $d \leq 3$ y $k \geq 1$. Sea $(\mathcal{T}_h)$ una familia regular de triangulaciones asociada a un d-símplice de tipo (k) o un d-paralelotopo de tipo (k). Entonces, el método de elementos finitos es convergente. Además, existe una constante C independiente de h tal que si la solución u pertenece a $H^{l+1}(\Omega)$, tenemos

$$\|u - u_h\|_{1,\Omega} \leq Ch^l |u|_{l+1,\Omega}.$$

Con las hipótesis del Teorema 5.61, teníamos un error de interpolación

$$\|u - \Pi_h u\|_{1,\Omega} = O(h^k)$$
$$\|u - \Pi_h u\|_{0,\Omega} = O(h^{k+1})$$

y hemos obtenido como error para el método de los elementos finitos

$$\|u - u_h\|_{1,\Omega} = O(h^k).$$

Introduzcamos el problema $(\mathcal{P}^\star)$. Dada una función g de $L^2(\Omega)$, buscamos φ solución en V de

$$\mathfrak{a}(v, \varphi) = \int_\Omega gv \, dx, \quad \forall v \in V. \tag{5.75}$$

Por el Teorema 5.18 (Lax-Milgram), el problema $(\mathcal{P}^\star)$ tiene una única solución.

Definición 5.63. *El problema $(\mathcal{P}^\star)$ se dice regular si la aplicación $g \mapsto \varphi$ es lineal y continua de $L^2(\Omega)$ en $H^2(\Omega)$: para todo $g \in L^2(\Omega)$, la solución φ de (5.75) pertenece a $H^2(\Omega) \cap V$ y existe una constante $C^\star > 0$ tal que*

$$\|\varphi\|_{2,\Omega} \leq C^\star \|g\|_{0,\Omega}. \tag{5.76}$$

Suponiendo el problema $(\mathcal{P}^\star)$ regular, podemos mejorar la estimación (5.73) probando que en la norma de $L^2(\Omega)$ el error es de orden $k+1$ y no solamente de orden k. Concretamente, se tiene

Teorema 5.64. *En las hipóteis del Teorema 5.61, supongamos además que el problema $(P^\star)$ es regular. Entonces, existe una constante C independiente de h tal que si la solución u de $(\mathcal{P})$ pertenece a $H^{k+1}(\Omega)$, tenemos*

$$\|u - u_h\|_{0,\Omega} \leq Ch^{k+1} |u|_{k+1,\Omega}. \tag{5.77}$$

Demostración. Escribimos

$$\|u - u_h\|_{0,\Omega} = \sup_{\substack{g \in L^2(\Omega) \\ g \neq 0}} \frac{\displaystyle\int_\Omega g(u - u_h) \, dx}{\|g\|_{0,\Omega}}.$$

Definimos φ como la solución en V de (5.75). Como el problema $(P^\star)$ es regular por hipótesis, tenemos (5.76). Como $u - u_h$ pertenece a V, tenemos por la construcción de φ

$$\int_\Omega g(u - u_h) \, dx = \mathfrak{a}(u - u_h, \varphi).$$

Por otra parte, como u y u_h son respectivamente soluciones de $(\mathcal{P})$ y de $(\mathcal{P}_h)$, tenemos

$$\mathfrak{a}(u - u_h, \varphi_h) = 0, \quad \forall \varphi_h \in V_h.$$

Deducimos

$$\int_\Omega g(u - u_h)\, dx = \mathfrak{a}(u - u_h, \varphi - \varphi_h)$$

para toda función φ_h de V_h, de donde

$$\int_\Omega g(u - u_h)\, dx \leq M\|u - u_h\|_{1,\Omega} \inf_{\varphi_h \in V_h} \|\varphi - \varphi_h\|_{1,\Omega}.$$

Como φ pertenece a $H^2(\Omega)$ y $d \leq 3$, podemos elegir $\varphi_h = \Pi_h\varphi$ y, aplicando teoría de interpolación, sabemos que existe una constante $C_1^\star$ que solo depende del elemento finito de referencia y de σ, tal que

$$\inf_{\varphi_h \in V_h} \|\varphi - \varphi_h\|_{1,\Omega} \leq \|\varphi - \Pi_h\varphi\|_{1,\Omega} \leq C_1^\star h|\varphi|_{2,\Omega}$$

y por (5.76)

$$\inf_{\varphi_h \in V_h} \|\varphi - \varphi_h\|_{1,\Omega} \leq C_2^\star h\|g\|_{0,\Omega},$$

con $C_2^\star = C_1^\star C^\star$. Obtenemos así

$$\int_\Omega g(u - u_h)\, dx \leq C_3^\star h\|u - u_h\|_{1,\Omega}\|g\|_{0,\Omega},$$

con $C_3^\star = MC_2^\star$. Deducimos

$$\|u - u_h\|_{0,\Omega} \leq C_3^\star h\|u - u_h\|_{1,\Omega}$$

y el resultado se deduce aplicando el Teorema 5.61.

6. Resolución de problemas de EDPs con Python

En este capítulo vamos a aproximar la solución de problemas de EDPs mediante el método de los elementos finitos que acabamos presentamos en el capítulo 5. En la resolución utilizaremos Python. Primero para un caso sencillo, realizaremos la programación del método y, posteriormente, utilizaremos el software FENICs que incorpora Python para la resolución de problemas de ecuaciones en derivadas parciales mediante elementos finitos.

6.1. Progación de una onda sobre una membrana cuadrada

Estamos interesados en resolver de forma aproximada el problema de ecuaciones en derivadas parciales

$$\begin{cases} \partial_{tt}^2 u - c^2\big(\partial_{xx}^2 u + \partial_{yy}^2 u\big) = 0 \ \text{ en } \Omega \times (0,T) \\ u(x,y,t) = 0 \ \text{ si } (x,y) \in \partial\Omega,\ t \in (0,T), \\ u(x,y,0) = u_0(x,y),\ \partial_t u(x,y,0) = v_0(x,y), \quad x \in \partial\Omega. \end{cases} \tag{6.1}$$

donde Ω es un conjunto abierto y acotado de $\mathbb{R}^2$, $T > 0$, $u_0 \in H_0^1(\Omega)$, $v_0 \in L^2(\Omega)$ $u_0 = 0$ sobre $\partial\Omega$.

Físicamente, el problema modela el movimiento de una membrana elástica. La variable $u(t,x,y)$ representa la altura de la partícula de la membrana en el instante t cuyas componentes horizontales son (x,y). La condición de contorno $u = 0$ sobre $\partial\Omega$ significa que la membrana está fija en el borde. Las funciones u_0 y v_0 nos dan la posición y la velocidad de la onda en el instante inicial. Por último, la constante c representa la velocidad de la onda.

Para resolver el problema usamos la formulación variacional que en este caso (que es un problema hiperbólico) se escribe como:

$$\begin{cases} u \in W^{1,\infty}(0,T;L^2(\Omega)) \cap L^\infty(0,T;H_0^1(\Omega)), \\ \dfrac{d^2}{dt^2}\displaystyle\int_\Omega u(x,y,t)w(x,y)\,dxdy + c^2\int_\Omega \nabla u(x,y,t)\cdot\nabla w(x,y)\,dxdy = 0 \text{ en } (0,T), \\ \forall\, w \in H_0^1(\Omega), \quad w = 0 \text{ sobre } \partial\Omega, \\ u(x,y,0) = u_0(x,y),\ \partial_t u(x,y,0) = v_0(x,y). \end{cases} \tag{6.2}$$

Para resolver este problema de forma numérica, seguimos el método de Galerkin que ya vimos para problemas elípticos en la sección 5.3. Sustituimos $H^1(\Omega)$ por un espacio finito-dimensional que llamamos V_r, siendo r su dimensión, formado por polinomios de grado k.

Si $\{\varphi_1, ..., \varphi_r\}$ es una base de V_r, tomamos las siguientes aproximaciones de u_0 y v_0:

$$u_0^r(x,y) = \sum_{i=1}^{r} u_{0,i}^r \varphi_i(x,y), \qquad v_0^r = \sum_{i=1}^{r} v_{0,i}^r \varphi_i(x,y). \tag{6.3}$$

El problema se convierte en encontrar

$$u^r(t,x,y) = \sum_{j=1}^{r} u_j^r(t)\varphi_j(x,y), \tag{6.4}$$

con u_j^r solución de

$$\sum_{j=1}^{r} k_{ij}\,(u_j^r)''(t) + c^2 \sum_{j=1}^{r} m_{ij}\,u_j^r(t) = 0, \quad i = 1, ..., r.$$

con

$$k_{ij} = \int_\Omega \varphi_i(x,y)\varphi_j(x,y)\,dxdy, \qquad m_{ij} = \int_\Omega \nabla\varphi_i(x,y)\cdot\nabla\varphi_j(x,y)\,dxdy, \tag{6.5}$$

o usando notación vectorial $U^r(t) = (u_1^r(t), ..., u_r^r(t))$ solución de

$$\begin{cases} K(U^r)''(t) + c^2 M U^r(t) = 0 \ \text{ en } [0,T],\ U^r(0) = U_0^r, \\ (U^r)'(0) = V_0^r,\ i = 1, ..., r, \end{cases} \tag{6.6}$$

con U_0^r, V_0^r los vectores de componentes $u_{0,i}^r$, $v_{0,i}^r$ y K, M las matrices de componentes k_{ij},m_{ij}. Estas matrices son simétricas y definidas positivas.

El problema (6.6) corresponde a una semidiscretización del problema dado que la variable t aún es continua. Para obtener una discretización total, usamos un método de resolución de un problema de valores iniciales para un sistema diferencial ordinario como estudiamos en el capítulo 3.

En nuestro caso, vamos a usar el método de Crank-Nicholson, dividiendo el intervalo $[0,T]$ en m partes iguales de anchura $h = T/m$. Llamamos $t_k = kh$, $k = 0, ..., m$ los puntos de la partición correspondiente. El esquema consiste en resolver de forma iterada

$$\begin{cases} \Big(\dfrac{4}{h^2}K + c^2 M\Big)U^{r,k+1} = \Big(\dfrac{4}{h^2}K - c^2 M\Big)U^{r,k} + \dfrac{4}{h}KV^{r,k}, \\ V^{r,k+1} = 2\dfrac{U^{r,k+1} - U^{r,k}}{h} - V^{r,k}. \end{cases} \tag{6.7}$$

Los vectores $U^{r,k} = (U_1^{r,k}, ..., U_r^{r,k})$, $V^{r,k} = (V_1^{r,k}, ..., V_r^{r,k})$ nos dan las aproximaciones de U^r y $\partial_t U^r$ en el tiempo t_k, i.e.

$$U^r(t_k,x) \sim \sum_{i=1}^{r} U_i^{r,k}\varphi_i(x), \quad V^r(t_k,x) \sim \sum_{i=1}^{r} V_i^{r,k}\varphi_i(x).$$

Se trata de un método iterado para calcular el valor de la onda en cada tiempo t_k resolviendo un sistema lineal de ecuaciones.

Como aproximación finito-dimensional del espacio $H^1(\Omega)$, usamos elementos finitos de Lagrange de grado 1. Consideramos una triangulación estructurada del dominio Ω, con n particiones en cada parte de la frontera y denotamos $\delta = \frac{1}{n}$.

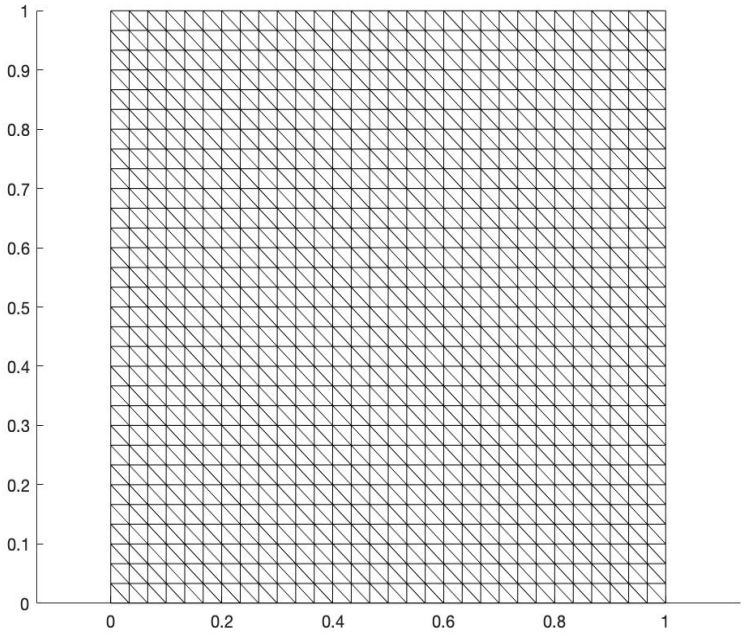

Figura 6.1: Triangulación para $n = 30$.

Como sabemos, una base está formada por las funciones que valen uno en un vértice y cero en los demás. Debido a la estructura particular del problema, resulta más simple usar matrices en lugar de vectores en el desarrollo anterior. Concretamente, en lugar de notar las funciones de base por φ_i, llamamos φ_{ij}, $1 \leq i, j \leq n-1$ a la función de $\mathbb{P}_1$ que vale uno en el vértice $(i\delta, j\delta)$ y cero en los demás.

Considerar estas funciones de base tiene la ventaja de que las funciones son de soporte reducido, lo que lleva a que las matrices que intervienen sean todas huecas. Desde el punto de vista informático, estas matrices son fáciles de incorporar y hay métodos numéricos específicos que simplifican los errores y los cálculos.

Una función cualquiera $w \in \mathbb{P}_1$ vendrá dada por

$$w(x, y) = \sum_{1 \leq i,j \leq n-1} w_{ij} \varphi_{ij}(x, y),$$

donde las componentes w_{ij} coinciden con los valores de w en los puntos $(i\delta, j\delta)$. De esta forma ahora tendremos funciones $U_{ij}^{r,k}, V_{ij}^{r,k}$ en lugar de $U_i^{r,k}$, $V_i^{r,k}$. Las matrices

K y M tendrán 4 índices (se suele hablar de tensores y no de matrices). Un cálculo explícito lleva a

$$k_{ijij} = \frac{\delta^2}{2}, \quad k_{iji-1j} = k_{iji+1j} = k_{ijij-1} = k_{ijij+1} = k_{iji-1j+1} = k_{iji+1j-1} = \frac{\delta^2}{12}.$$

Los demás elementos k_{ijlk} son nulos.

$$m_{ijij} = 4, \quad m_{iji-1j} = m_{iji+1j} = m_{ijij-1} = m_{ijij+1} = -1.$$

Los demás elementos m_{ijkl} son nulos.

Para programar el método no se necesita guardar en memoria K, M. Tan solo necesitamos una subrutina que nos diga como se multiplica Kw o Mw.

Si $w = (w_{ij})$ se tiene

$$(Kw)_{ij} = \frac{\delta^2}{12}\Big(6w_{ij}+w_{i-1j}+w_{i+1j}+w_{ij-1}+w_{ij+1}+w_{i-1j+1}+w_{i+1j-1}\Big) \text{ si } i,j \notin \{0,n\}.$$

En otro caso $(Kw)_{0i} = (Kw)_{i0} = (Kw)_{ni} = (Kw)_{in} = 0$.

Este es el código correspondiente a esta parte (numpy y matplotlib están importados al igual que en el capítulo 4):

```
def multK(u,n):
    r = 1/(12*n**2)
    Ku = 6 * u
    Ku[1:n+1,0:n+1] = Ku[1:n+1,0:n+1] + u[0:n,0:n+1]
    Ku[0:n,0:n+1] = Ku[0:n,0:n+1] + u[1:n+1,0:n+1]
    Ku[0:n+1,1:n+1] = Ku[0:n+1,1:n+1] + u[0:n+1,0:n]
    Ku[0:n+1,0:n] = Ku[0:n+1,0:n] + u[0:n+1,1:n+1]
    Ku[1:n+1,0:n] = Ku[1:n+1,0:n] + u[0:n,1:n+1]
    Ku[0:n,1:n+1] = Ku[0:n,1:n+1] + u[1:n+1,0:n]
    Ku = r*Ku
    return Ku
```

Análogamente

$$(Mw)_{ij} = 4w_{ij} - w_{i-1j} - w_{i+1j} - w_{ij-1} - v_{ijij+1}$$

y $(Mw)_{0i} = (Mw)_{i0} = (Mw)_{ni} = (Mw)_{in} = 0$.

```
def multM(u,n):
    Mu = 4 * u
    Mu[1:n+1,1:n+1] = Mu[1:n+1,1:n+1] - u[0:n,1:n+1]
    Mu[0:n,0:n+1] = Mu[0:n,0:n+1] - u[1:n+1,0:n+1]
    Mu[0:n+1,1:n+1] = Mu[0:n+1,1:n+1] - u[0:n+1,0:n]
    Mu[0:n+1,0:n] = Mu[0:n+1,0:n] - u[0:n+1,1:n+1]
    return Mu
```

Para resolver los sistemas lineales en (6.7) usamos el método del gradiente conjugado. En particular, obtendremos con este método los $U^{r,k}$ y a partir de estos los $V^{r,k}$, ya que vienen dados por una expresión explícita. Se trata de un método iterado para resolver un sistema de la forma $Ax = b$ con A una matriz simétrica definida positiva. Por tanto, el primer paso es obtener el segundo miembro, b, para nuestro problema. De esto se encarga la siguiente función:

```
def miemb2(c2,n,m,u,v):
    b = 4*m*multK (m*u+v,n) - c2 * multM(u,n)
    return b
```

El algoritmo es como sigue:

Se fija $\varepsilon > 0$ pequeño que representa el error máximo que queremos cometer.

- Tomamos una primera aproximación x_0 y definimos $r_0 = b - Ax_0$, $p_0 = r_0$.
- Supongamos calculados x_j, r_j, p_j. Si $\|r_j\|^2 < \varepsilon^2$ paramos y en otro caso seguimos iterando
- Tomamos

$$\lambda_j = \frac{\|r_j\|^2}{p_j^t A p_j}, \quad x_{j+1} = x_j + \lambda_j p_j, \quad r_{j+1} = r_j - \lambda_j A p_j, \quad p_{j+1} = r_{j+1} + \frac{\|r_{j+1}\|^2}{\|r_j\|^2} p_j.$$

En el algoritmo, $r = b - Ax$ es el residuo en el punto x. Si $\|r\| \leq \varepsilon$, con ε pequeño, entonces x es próximo a la solución del problema, es por ello que podemos utilizar r como una estimación del error y, por tanto, lo hemos tomado como condición de parada.

Observamos que tan solo necesitamos conocer el resultado de multiplicar A por un vector. Dicho producto se realiza con la función

```
def multA(c2,n,m,u):
    Au = 4*m**2*multK (u,n) + c2 * multM(u,n)
    return Au
```

No obstante, las condiciones de contorno nos exigen que $u = 0$ sobre la frontera, luego utilizamos el siguiente método para corregir este problema. En la práctica, es usual tomar condiciones de Neumann a modo de penalización.

```
def cfront(u,n):
    w = u
    w[0,0:n+1] = np.zeros((1,n+1))
    w[n,0:n+1] = np.zeros((1,n+1))
    w[1:n,0] = np.zeros(n-1)
    w[1:n,n] = np.zeros(n-1)
    return w
```

Como el sistema que hay que resolver para obtener $U^{r,k+1}$ es muy parecido al que resuelve $U^{r,k}$ se utiliza $U^{r,k}$ como inicialización para resolver el sistema para $U^{r,k+1}$.

De este modo, los sistemas se resuelven muy rápidamente. Aquí mostramos el código correspondiente al método descrito:

```
def gradCon(b,u0,c2,m,n):
#metodo del gradiente conjugado para nuestro sistema lineal
    ep = 10**-12
    u = u0
    r = b - multA(c2,n,m,u)
    r = cfront(r,n)
    p = r
    #tambien es posible n2r = np.linalg.norm(r)**2 pero
       produce errores al redondear la raiz cuadrada
    n2r = np.sum(np.multiply(r,r))
    while n2r > ep:
        Ap = multA(c2,n,m,p)
        Ap = cfront(Ap,n)
        pAp = np.sum(np.multiply(Ap,p))
        lam = n2r/pAp
        u = u + lam*p
        r = r - lam*Ap
        n2ra = n2r
        n2r = np.sum(np.multiply(r,r))
        p = r + n2r/n2ra * p
    return u
```

Igual que hicimos en el capítulo 4, vamos a concluir mostrando dos animaciones a modo de simulación del movimiento de la membrana.En la primera, consideramos un impulso en el centro y en la segunda en las esquinas. Este es el código que las genera:

```
def ondas(c,n,h,v,j):

    c2 = c**2
    m = 1/h
    x=np.linspace(0,1,n+1)
    y=x
    #intervalo de tiempo
    [X,Y]=np.meshgrid(x,y)
    u=np.zeros((n+1,n+1))
    #funcion inicial (es la alpha)
    u0 = u

    lu=[u]
    for k in range(0,500):
        ua = u
        #obtenemos el 2 miembro de la ecuacion
        b = miemb2(c2,n,m,u,v)
        u = gradCon(b,u0,c2,m,n)
```

```
        lu.append(u)
        u0 = u
        v = 2*m*(u-ua) - v
    fig=plt.figure()
    ax=fig.add_subplot(projection = '3d')
    def actualizar(i):
        print(i)
        ax.clear()
        surf=ax.plot_surface(X,Y,lu[i],cmap='viridis')
        #establecemos los ejes
        ax.set_xlim(0,1)
        ax.set_ylim(0,1)
        ax.set_zlim(-0.5,0.5)
    #generamos la animacion y la almacenamos en video
    writervideo = animation.FFMpegWriter(fps=20)
    anim=animation.FuncAnimation(fig,actualizar,range
       (0,500),interval=1,repeat=False,blit=False)
    anim.save('onda'+j+'.mp4', writer=writervideo)
    return anim
```

Por último, mostramos algunos instantes de la segunda animación generada del siguiente modo:

```
#c es la velocidad de la onda
c = 1
#numero de particiones del cuadrado
n = 150
h = 0.01
#velocidad inicial. Hay que hacer cambios porque si no la
   onda no se mueve
v1 = np.zeros((n+1,n+1))
#Por ejemplo, le damos un golpe al centro del cuadrado; le
   damos velocidad
v1[4,4:145]=50*np.ones((1,141))
#ondas(c,n,h,v1,'1')
v2=np.zeros((n+1,n+1))
v2[5,5]=500
v2[145,145]=-500
ondas(c,n,h,v1,'1')
ondas(c,n,h,v2,'2')
```

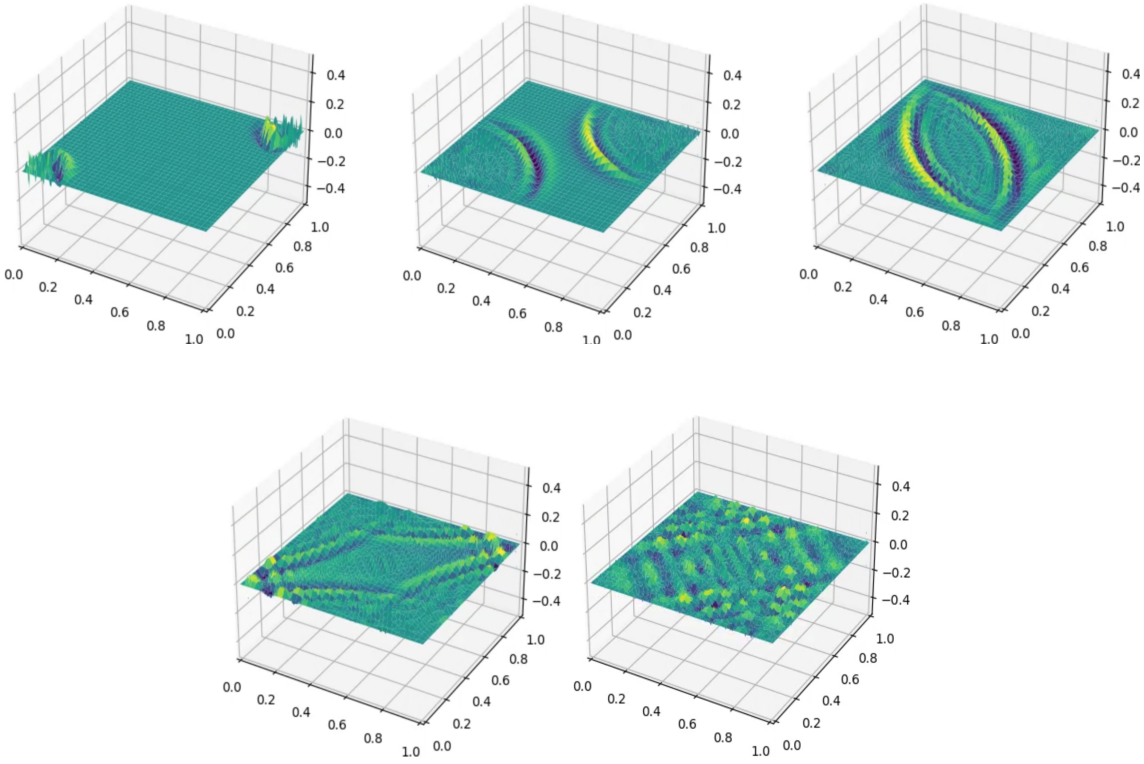

Figura 6.2: Fotogramas pertenecientes a la animación generada para la propagación de la onda.

6.2. Resolución de problemas con FEniCS

En esta sección vamos a resolver algunos problemas de EDPs utilizando el software ya incorporado en Python: FEniCS. Para la representación gráfica, hemos utilizado Python y ParaView. Este último es un software de visualización y análisis de datos de código abierto y gratuito para Gnu/Linux, Windows y MacOS.

6.2.1. Preparación del entorno

Comenzamos indicando cómo utilizar FEniCS en nuestro equipo. Para ello, nos basamos en las instrucciones dadas en [8]. Sin embargo, no son muy extensas y el libro se publicó en 2016, por lo que hay elementos del software expuesto que han quedado obsoletos. Por tanto, mostramos una guía actualizada para Windows10/11, Linux y MacOSX:

Windows10/11

En primer lugar, hay que comentar que este software solo está disponible para equipos con Sistema Operativo Linux, por lo que no podemos instalar este módulo de forma directa.Tal y como se indica en la documentación oficial de FEniCS, que

puede encontrarse en https://fenicsproject.org/, es posible trabajar con FEniCS en Windows mediante el uso de contenedores de Docker, ya que existen algunos debidamente configurados para trabajar con FEniCS. Estos son los pasos a seguir:

1. Descargar la versión más reciente de la aplicación de Ubuntu de la Microsoft Store. Esto habilita el subsistema de Linux para Windows, lo que nos permite abrir una terminal Ubuntu en nuestro escritorio Windows.
2. Abrir la aplicación e introducir un nombre de usuario y una contraseña para nuestra terminal de Linux.
3. Descargar Docker: https://www.docker.com/
4. Ejecutar el comando siguiente en una terminal de Ubuntu: docker run -ti -p 127.0.0.1:8000:8000 -v $(pwd):/home/fenics/shared -w /home/fenics/shared quay.io/fenicsproject/stable:current. Esto nos proporciona un entorno con este software debidamente configurado
5. Utilizar el comando sudo nano nombredearchivo.py y editarlo con el contenido del programa en cuestión.
6. Introducir el comando python3 nombredearchivo para ejecutar el programa correspondiente. En caso de que genere ficheros, podrán copiarse a nuestro sistema Windows mediante el comando cp path_archivo_origen path_archivo_destino y teniendo en cuenta que el directorio Linux vendrá dado por $\sim$ y el de Windows por /.

Sin embargo, esta instalación tiene el problema que, a la hora de realizar representaciones gráficas para comprobar la corrección de los resultados obtenidos, se requiere estar copiando archivos de la terminal de Linux a Windows continuamente.

Una alternativa para resolver este inconveniente es utilizar una máquina virtual de Ubuntu en nuestro equipo. Para ello:

1. Descargar en el sistema operativo anfitrión algún visor de máquinas virtuales en caso de no tener uno ya instalado. En particular, se ha escogido Oracle VM VirtualBox, que puede encontrarse en https://www.virtualbox.org/. Si ya había un visor instalado, se recomienda actualizarlo a la versión más reciente.

   Por ejemplo, la versión 6.1 de VirtualBox es incompatible con la distribución 22.04 de Ubuntu.
2. Descargar el disco ISO correspondiente a la distribución de Linux deseada. Este es el enlace de descarga del disco ISO de Ubuntu 22.04: https://releases.ubuntu.com/jammy/
3. Crear una máquina virtual en el visor a partir del ISO anterior. Se nos ofrecerá la posibilidad de instalar Ubuntu de forma automática, pero internamente se instalará en modo protegido de modo que no podremos utilizar la terminal de comandos en el sistema Ubuntu resultante. Por esto, se recomienda la configuración manual siguiendo esta guía: https://ubuntu.com/tutorials/

install-ubuntu-desktop#1-overview

4. Configurar FEniCS siguiendo las instrucciones que a continuación se detallan para Ubuntu.

Ubuntu

En un sistema Ubuntu, ya sea anfitrión o una máquina virtual, configuraremos FEniCS del siguiente modo:

1. Instalar Python en nuestra máquina virtual. Aunque la versión 3.11 es la más actual, es incompatible con FEniCS, por lo que recomendamos la 3.10 o anteriores.
2. Instalar un entorno de programación, por ejemplo, Visual Studio, así como el visualizador ParaView para las representaciones gráficas.
3. Instalar la API de Fenics mediante los siguientes comandos en una terminal:
   a) sudo add-apt-repository ppa:fenics-packages/fenics
   b) sudo apt-get update
   c) sudo apt-get install fenics
   d) sudo apt-get dist-upgrade
4. Comprobar la instalación de Fenics con el comando python -c 'import fenics'
5. Instalar los paquetes necesarios de Python para trabajar. Por ejemplo, en este trabajo se han instalado matplotlib y numpy con los comandos: python3 -m pip install numpy, python3 -m pip install matplotlib.
6. Instalar FFmpeg para poder guardar animaciones con: sudo apt-install ffmpeg.

MacOSX

En el caso de MacOSX, la opción más sencilla es descargar el programa Anaconda y e instalar los paquetes de Python requeridos en la terminal que nos ofrece este entorno software. La misma opción es válida para Linux. Sin embargo, la distribución de Anaconda para Windows no tiene FEniCS disponible.

6.2.2. Ensayos numéricos

En esta sección mostraremos los diferentes problemas de EDPs que hemos resuelto en la presente memoria utilizando la biblioteca FEniCS. Por simplificar, supondremos importados dolfin, matplotlib, mshr (módulo que nos permite construir entornos geométricos de forma sencilla), ufl (para poder utilizar el operador nabla_grad), numpy, matplotlib.pytplot y matplotlib.animation. Por tanto, supondremos que todos los ficheros.py tienen la cabecera:

```
from dolfin import*
from mshr import *
from ufl import nabla_div
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
```

Problema de Poisson

Como primer problema y a fin de mostrar los conceptos básicos para el uso de la biblioteca FEniCS, hemos escogido un problema muy simple. Se trata de resolver la ecuación de Poisson en el disco unidad con condiciones de Dirichlet homogéneas y segundo miembro constante, i.e.

$$\begin{cases} -\Delta u = 1 \ \text{ en } B(0,1) \\ u = 0 \ \text{ sobre } \partial B(0,1), \end{cases}$$

cuya formulación variacional es

$$\begin{cases} u \in H_0^1(B(0,1)) \\ \displaystyle\int_{B(0,1)} \nabla u \cdot \nabla v \, dx = \int_{B(0,1)} v \, dx, \quad \forall v \in H_0^1(B(0,1)). \end{cases}$$

La solución de este problema se puede obtener fácilmente pasando a componentes radiales y viene dada por

$$u(x) = \frac{1 - |x|^2}{4}, \quad \forall x \in B(0,1).$$

El hecho de que el problema es sencillo y que conocemos la solución exacta nos permite comprobar algunos de los resultados teóricos presentados en el capítulo 5 relativos a los elementos finitos. Concretamente, vamos a validar los órdenes de convergencia.

Siguiendo el mismo proceso desarrollado en el capítulo 4, tomamos distintos pasos de malla h y vamos a comprobar que el método tiene una convergencia de orden 1 en $H^1(\Omega)$ y 2 en $L^2(\Omega)$ para elementos finitos $\mathbb{P}_1$.

Procedemos a describir el código implementado para una malla concreta. En primer lugar, indicamos el dominio y la triangulación que vamos a realizar mediante las siguientes instrucciones:

```
#figura
domain=Circle(Point(0,0),1)
#triangulacion
mesh=generate_mesh(domain,80)
```

Cabe destacar que el haber puesto el parámetro 20, se tendrá que la talla de los triángulos será del orden del diámetro del disco, 2, entre 20. A continuación, definimos el espacio sobre el que vamos a trabajar:

```
#espacio
V=FunctionSpace(mesh,'P',1)
```

El siguiente paso es definir las condiciones de contorno:

```
#condicion de contorno
u_B=Expression('0', degree=2)
def boundary(x, on_boundary):
    return on_boundary
bc=DirichletBC(V,u_B, boundary)
```

El método boundary nos permite definir la frontera de nuestro problema de modo que devolverá cierto si un punto está dentro de la misma y falso en caso contrario; la variable u_B el valor de la solución en la misma. Por último, el método DirichletBC nos permite definir estas condiciones en el espacio correspondiente. Ahora vamos a definir la formulación variacional de nuestro problema:

```
#formulacion variacional
u=TrialFunction(V)
v=TestFunction(V)
f=Constant(1)
a=dot(grad(u),grad(v))*dx
L=f*v*dx
u=Function(V)
```

Los métodos TrialFunction() y TestFunction() nos permiten definir la función incógnita u en la forma bilineal y la función v test en las formas bilineal y lineal, respectivamente. Asimismo, hay que comentar que el método dot(.,.) nos permite realizar el producto escalar entre dos vectores (en este caso los gradientes de u y v) y que multiplicar por dx significa integrar en el disco. Se define u como una función genérica del espacio V para poder albergar la solución que calculamos a continuación:

```
#resolvemos
solve(a==L,u,bc)
```

En este problema conocemos la solución, que indicaremos por

```
u_D=Expression('(1-x[0]*x[0]-x[1]*x[1])*0.25',degree=2)
```

Podemos calcular el error de la solución de la siguiente forma:

```
error=errornorm(u_D,u,'L2')
print('err u: ',error)
```

Para el cálculo del error del gradiente, creamos un espacio W de funciones vectoriales de orden $\mathbb{P}_0$.

```
u_D=project(u_D,V)
L2cuad=dot(grad(u-u_D),grad(u-u_D))*dx
error_grad=sqrt(assemble(L2cuad))
print('err grad: ',error_grad)
```

Dado que por defecto dot(.,.)*dx pertenece a una clase de funciones interna de la librería que sirve de argumento a la formulación variacional, necesitamos el método assemble() para traducir de dicha clase a su valor numérico.

Tras realizar diversas pruebas, obtenemos los siguientes resultados.Para los órdenes, hemos usado la misma fórmula aproximada que en el capítulo 4

h	Error	Orden: $\log_{\frac{h_{anterior}}{h_{actual}}}(\frac{err_{anterior}}{err_{actual}})$
0,4	0,015349645261718803	
0,2	0,00415834813667352	1,884122863835053
0,1	0,0010001305863290004	2,0558221616513364
0,05	0,00024799102983962624	2,011828541450095
0,025	$6{,}227513325964079e-05$	1,9935598286490113

Cuadro 6.1: Tabla con los errores cometidos para la solución aproximada.

h	Error grad.	Orden grad.: $\log_{\frac{h_{anterior}}{h_{actual}}}(\frac{err_{anterior}}{err_{actual}})$
0,4	0,017704326821673288	
0,2	0,017064088851406357	0,05313860581806488
0,1	0,005495577725145336	1,6346203245717297
0,05	0,0023432505785322636	1,229759914217879
0,025	0,001244185434341348	0,9133097170539751

Cuadro 6.2: Tabla con los errores cometidos para el gradiente de la solución aproximada.

Luego queda validado el método. Finalizamos este problema mostrando una representación gráfica en ParaView de la solución obtenida por el método. Para ello ejecutamos:

```
VTKFile=File('poisson/solutioncirc.pvd')
VTKFile << u
```

Esto nos genera un archivo en formato VTK, que es apto para ParaView. Una vez abierto en dicho programa, hemos usado el filtro WarpByScalar para obtener el gráfico de la solución en 3D. Asimismo, también se ha representado con la triangulación marcada, mediante la opción de visualización Surface With Edges.

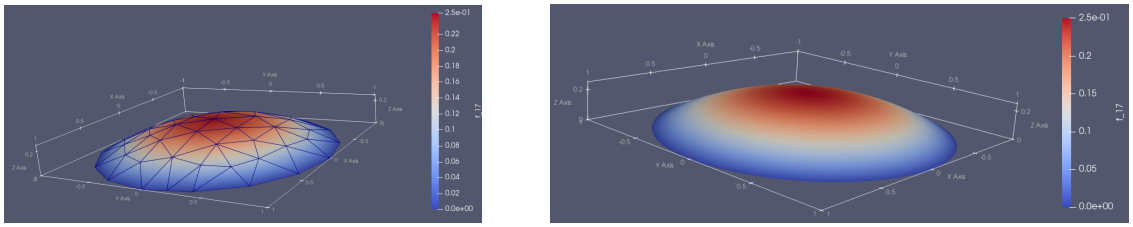

Figura 6.3: Representación en ParaView de la solución con y sin mallado.

Cálculo del primer autovalor del Laplaciano.

Vamos a usar el método de la potencia para calcular el primer autovalor λ del operador Laplaciano en un dominio acotado Ω con condiciones de Dirichlet homogéneas, es decir, el valor λ más pequeño tal que existe una función u no identicamente nula solución de

$$\begin{cases} -\Delta u = \lambda u \ \text{ en } \Omega \\ u = 0 \ \text{ sobre } \partial\Omega. \end{cases} \tag{6.8}$$

Se sabe que este valor λ existe y es positivo y el espacio de autovectores correspondiente, soluciones de (6.8), es un espacio de dimensión 1 generado por una función que es estrictamente positiva en Ω.

Este valor tiene una gran importancia en muchos problemas de física, biología, etc. Por ejemplo, su inversa nos da la mejor constante de Poincaré y se conoce que la solución de la ecuación del calor con un segundo miembro independiente del tiempo converge a la solución del problema estacionario con velocidad $e^{-\lambda t}$.

Para calcular el primer autovalor, vamos a usar el método de la potencia normalizado. El Algoritmo es el siguiente:

1. Inicialización:
   - Se toma $v_0 \in H_0^1(\Omega)$, positiva, no nula.
   - Se calculan
   $$a_0 = \int_\Omega |v_0|^2 dx, \quad b_0 = \int_\Omega |\nabla v_0|^2 dx.$$
   - Se toma
   $$u_0 = \frac{v_0}{\sqrt{a_0}}, \quad \lambda_0 = \frac{b_0}{a_0}.$$
2. Recurrencia
   - Se toma v_{n+1} solución de
   $$\begin{cases} -\Delta v_{n+1} = u_n \ \text{ en } \Omega \\ v_{n+1} = 0 \ \text{ sobre } \partial\Omega \end{cases}$$

- Se calculan

$$a_{n+1} = \int_\Omega |v_{n+1}|^2 dx, \quad b_{n+1} = \int_\Omega |\nabla v_{n+1}|^2 dx.$$

- Se toma

$$u_{n+1} = \frac{v_{n+1}}{\sqrt{a_{n+1}}}, \quad \lambda_{n+1} = \frac{b_{n+1}}{a_{n+1}}.$$

Como ejemplo vamos a considerar el caso simple $\Omega = (0, \pi)^2$ con

$$\lambda = 2, \quad u = \frac{2}{\pi} \sin(x) \sin(y).$$

Como inicialización tomamos

$$v_0(x, y) = x(\pi - x)y(\pi - y).$$

Introducimos una tolerancia de 10^{-4} como test de parada. Hemos obtenido la aproximación $\lambda = 2{,}0000157520$. Hay que tener en cuenta que el método no va a converger a 2 exactamente, sino al primer autovalor de la aproximación discreta realizada por FEniCS, en la que se comete un error de truncamiento.

Los tests se han realizado con el siguiente método:

```
def lapval(tol):
```

En primer lugar, almacenamos la solución y el valor del primer autovalor asociado al problema para poder realizar las mediciones de error.

```
    #valores reales
    lad=2
    ud=Expression('2/pi*sin(x[0])*sin(x[1])',degree=2)
```

A continuación, realizamos la etapa de inicialización

```
    #Inicializacion
    w=Expression('x[0]*(pi-x[0])*x[1]*(pi-x[1])',degree=4)
    #figura
    domain=Rectangle(Point(0,0),Point(np.pi,np.pi))
    #triangulacion
    mesh=generate_mesh(domain,80) #h=pi/40 aprox.
    #espacio
    V=FunctionSpace(mesh,'P',1)
    ud=project(ud,V)
    w=project(w,V)
    a=assemble(w*w*dx)
    b=assemble(dot(grad(w),grad(w))*dx)
    u=w/sqrt(a)
    la=b/a
    err_la=np.abs(la-lad)
```

Definimos las condiciones de contorno:

```
#condiciones de Dirichlet
u_D=Expression('0', degree=2)
def boundary(x, on_boundary):
        return on_boundary
bc=DirichletBC(V,u_D, boundary)
```

Iteramos realizando la fase que hemos denominado recurrencia.

```
i=0
while err_la>tol and i<100:
    w=TrialFunction(V)
    v=TestFunction(V)
    A=dot(grad(w),grad(v))*dx
    L=u*v*dx
    w=Function(V)
    solve(A==L,w,bc)
    a=assemble(w*w*dx)
    b=assemble(dot(grad(w),grad(w))*dx)
    u=w/sqrt(a)
    u=project(u,V)
    la=b/a
```

Asimismo, en cada paso calculamos los errores cometidos para λ, u y ∇u, así como los mostramos por pantalla. Finalmente, devolvemos el λ obtenido.

```
    err_la=np.abs(la-lad)
    error=errornorm(ud,u,'L2')
    L2cuad=dot(grad(u-ud),grad(u-ud))*dx
    err_grad=sqrt(assemble(L2cuad))
    print("================================")
    print("ITERACION ",i,':')
    print('Para lambda: ',err_la)
    print('Para u: ',error)
    print('Para grad(u): ',err_grad)
    i=i+1
return la
```

Deformación elástica de una placa en 3D.

En este ejemplo, consideramos una placa elástica definida por el prisma cuadrangular $\Omega = (0,7) \times (0,7) \times (0,1)$ sometida a la fuerza de la gravedad y sujeta por un pilar que la atraviesa en el prisma $C = (3,4) \times (3,4) \times (0,1)$. Desde el punto de vista físico, el problema viene modelado por:

$$\begin{cases} -2\mu \operatorname{div} e(u) - \lambda \nabla \operatorname{div} u = -ge_3 \ \text{ en } \Omega \setminus C, \\ u = 0 \ \text{ sobre } \partial C, \qquad 2\mu e(u)\nu + \lambda \operatorname{div} u\, \nu = 0 \ \text{ sobre } \partial\Omega \setminus \partial C, \end{cases}$$

donde e_3 es el tercer vector de la base canónica, μ y λ las constantes de Lamé del material elástico, ν la normal unitaria exterior a Ω, $e(u)$ la parte simétrica de la derivada que en elasticidad se suele conocer como tensor de deformaciones. La función vectorial u solución del problema representa la deformación de la placa. Esto significa que un punto que en ausencia de fuerzas se encontrara en la posición x, pasará a ocupar la posición $x + u(x)$ después de la deformación.

En nuestro experimento numérico hemos escogido valores académicos dados por: $\mu = 1$, $\lambda = 1{,}25$, $g = 0{,}02$.

```
#Constantes
mu=1
lam=1.25
g=0.02
```

Escribimos nuestro dominio como diferencia de los dos conjuntos mencionados y generamos la triangulación:

```
#Dominio
domain=Box(Point(0,0,0),Point(7,7,1))-Box(Point(3,3,0),
   Point(4,4,1))
#Triangulacion
mesh=generate_mesh(domain,40)
```

Definimos el espacio y las condiciones de contorno teniendo en cuenta los trozos de frontera existentes. Debido a las imprecisiones cometidas por FEniCS, es recomendable introducir una tolerancia en lugar de usar los operadores $\geq$ y $\leq$ para definir los recintos.

```
#Espacio
V=VectorFunctionSpace(mesh,'P',1)
#Definimos las condiciones de contorno
tol= 1E-14
def boundaries(x,on_boundary):
    return on_boundary and ((x[0]>3-tol and x[0]<3+tol) or
                            (x[0]>4-tol and x[0]<4+tol) or
                            (x[1]>3-tol and x[1]<3+tol) or
                            (x[1]>4-tol and x[1]<4+tol))
bc = DirichletBC(V,Constant((0,0,0)),boundaries)
```

Introducimos tanto el tensor de deformaciones como el de esfuerzos:

```
#Tensor de deformaciones
def epsilon(u):
    return 0.5*(nabla_grad(u)+nabla_grad(u).T)
#Tensor esfuerzos
def sigma(u):
    return lam*nabla_div(u)*Identity(d)+2*mu*epsilon(u)
```

A continuación, indicamos el problema variacional. Destaca el método inner(.,.), que se encarga de realizar productos entre matrices.

```
#Definimos el problema variacional
u=TrialFunction(V)
d=u.geometric_dimension()
v=TestFunction(V)
f=Constant((0,0,-g))
T=Constant((0,0,0))
a=inner(sigma(u),epsilon(v))*dx
L=dot(f,v)*dx+dot(T,v)*ds
```

Resolvemos el problema y generamos un archivo en formato VTK con el estado inicial de la placa para su posterior visualización en ParaView.

```
#Lo resolvemos
u=Function(V)
solve(a==L,u,bc)
VTKFile=File('inicial.pvd')
VTKFile << u
```

Esta es la representación obtenida:

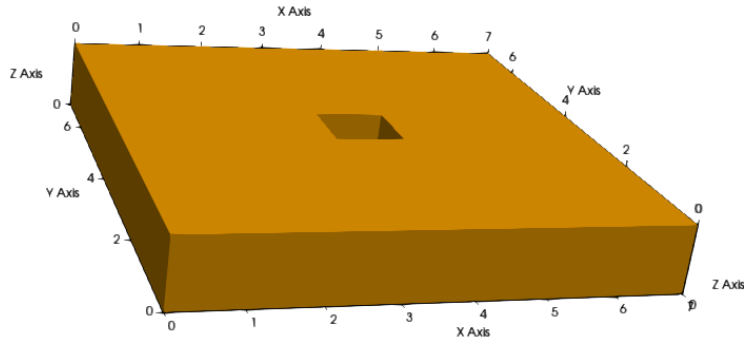

Figura 6.4: Representación en ParaView del estado inicial de la placa.

Por último, dibujamos cara a cara la deformación producida utilizando matplotlib por su mayor versatilidad. Dado que en el prisma cuadrangular hueco no está definida la u obtenida, mediante el siguiente método podemos evaluar u en todo Ω.

```
def Wesc(x,y,z):
    if x>=3 and x<=4 and y>=3 and y<=4:
        w=np.array([0,0,0])
    else:
        w=u(x,y,z)
    return w
```

Dibujamos a la vez las caras pertenecientes a planos paralelos al plano OXY:

```
x=np.linspace(0,7,700)
y=np.linspace(0,7,700)
X,Y=np.meshgrid(x,y)
fig=plt.figure()
ax=fig.add_subplot(projection='3d')
A1=np.zeros((len(x),len(y)))
A2=np.zeros((len(x),len(y)))
A3=np.zeros((len(x),len(y)))
B1=np.zeros((len(x),len(y)))
B2=np.zeros((len(x),len(y)))
B3=np.zeros((len(x),len(y)))

for i in range(0,len(x)):
    for j in range(0,len(y)):
        A1[i,j]=X[i,j]+Wesc(X[i,j],Y[i,j],0)[0]
        A2[i,j]=Y[i,j]+Wesc(X[i,j],Y[i,j],0)[1]
        A3[i,j]=Wesc(X[i,j],Y[i,j],0)[2]
        B1[i,j]=X[i,j]+Wesc(X[i,j],Y[i,j],1)[0]
        B2[i,j]=Y[i,j]+Wesc(X[i,j],Y[i,j],1)[1]
        B3[i,j]=1+Wesc(X[i,j],Y[i,j],1)[2]
ax.plot_surface(A1,A2,A3,color="orange")
ax.plot_surface(B1,B2,B3,color="orange")
```

Caras pertenecientes a planos paralelos al plano OYZ:

```
z=np.linspace(0,1,100)
Y,Z=np.meshgrid(y,z)
C1=np.zeros((len(z),len(y)))
C2=np.zeros((len(z),len(y)))
C3=np.zeros((len(z),len(y)))
D1=np.zeros((len(z),len(y)))
D2=np.zeros((len(z),len(y)))
D3=np.zeros((len(z),len(y)))
```

Caras pertenecientes a planos paralelos al plano OXZ:

```
X,Z2=np.meshgrid(x,z)
E1=np.zeros((len(z),len(x)))
E2=np.zeros((len(z),len(x)))
E3=np.zeros((len(z),len(x)))
F1=np.zeros((len(z),len(x)))
F2=np.zeros((len(z),len(x)))
F3=np.zeros((len(z),len(x)))
```

Dado que la longitud de las variables x e y coinciden, podemos representar estas mismas caras modificando los valores de las matrices en el mismo bucle:

```
for i in range(0,len(z)):
    for j in range (0,len(y)):
```

```
C1[i,j]=Wesc(0,Y[i,j],Z[i,j])[0]
C2[i,j]=Y[i,j]+Wesc(0,Y[i,j],Z[i,j])[1]
C3[i,j]=Z[i,j]+Wesc(0,Y[i,j],Z[i,j])[2]
D1[i,j]=7+Wesc(7,Y[i,j],Z[i,j])[0]
D2[i,j]=Y[i,j]+Wesc(7,Y[i,j],Z[i,j])[1]
D3[i,j]=Z[i,j]+Wesc(7,Y[i,j],Z[i,j])[2]

E1[i,j]=X[i,j]+Wesc(X[i,j],0,Z2[i,j])[0]
E2[i,j]=Wesc(X[i,j],0,Z2[i,j])[1]
E3[i,j]=Z2[i,j]+Wesc(X[i,j],0,Z2[i,j])[2]
F1[i,j]=X[i,j]+Wesc(X[i,j],7,Z2[i,j])[0]
F2[i,j]=7+Wesc(X[i,j],7,Z2[i,j])[1]
F3[i,j]=Z2[i,j]+Wesc(X[i,j],7,Z2[i,j])[2]
```

Dibujamos las 4 caras:

```
ax.plot_surface(C1,C2,C3,color="orange")
ax.plot_surface(D1,D2,D3,color="orange")
ax.plot_surface(E1,E2,E3,color="orange")
ax.plot_surface(F1,F2,F3,color="orange")
```

Finalizamos estableciendo los ejes y mostrando la representación obtenida:

```
ax.set_xlim(-1,8)
ax.set_ylim(-1,8)
ax.set_zlim(-1,8)
plt.show()
```

Esta es la deformación de la placa resultante. Se observa que la parte atravesada por la hipotética viga no presenta deformación alguna. Esta es la representación obtenida:

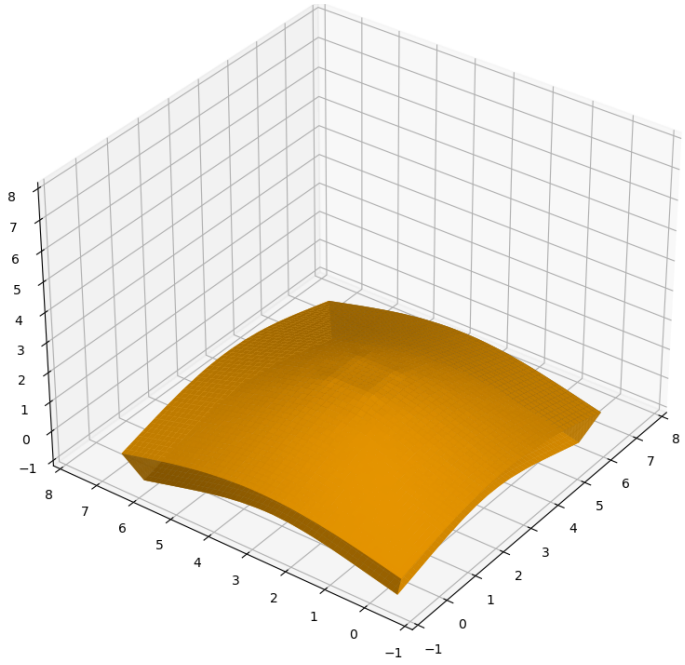

Figura 6.5: Representación con matplotlib de la placa deformada.

Difusión del calor entre dos recintos conectados por una interfaz

En este problema, consideramos un conjunto abierto en $\mathbb{R}^2$ con dos componentes conexas dadas por $\Omega_1 = (-1, 0) \times (0, 1)$ y $\Omega_2 = (0, 1) \times (0, 1)$ que, desde el punto de vista de la aplicación, representarían dos habitaciones. Suponemos que en el tiempo inicial la temperatura de ambas habitaciones, así como la temperatura del exterior es nula. En ese momento, en el dominio Ω_2, se sitúa una fuente de calor constante en el conjunto $C = (0{,}4, 0{,}6)^2$. El problema es estudiar cómo se difunde el calor y, en particular, cómo pasa a la otra habitación a través de la pared en cómun $\Gamma = \{0\} \times (0, 1)$. La figura 6.6 ilustra la situación descrita.

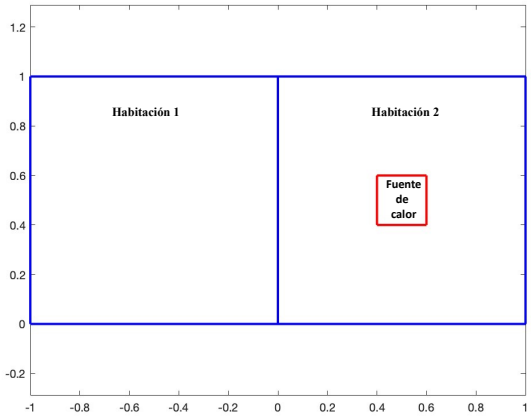

Figura 6.6: Ilustración de los recintos a estudiar.

Desde el punto de vista físico, el problema viene modelado por:

$$\begin{cases} \partial_t u_1 - \mu\Delta u_1 = 0 \ \text{ en } \Omega_1, \\ \dfrac{\partial u_1}{\partial \nu_1} + k u_1 = 0 \ \text{ sobre } \partial\Omega_1 \setminus \Gamma, \quad \dfrac{\partial u_1}{\partial \nu_1} + k(u_1 - u_2) = 0 \ \text{ sobre } \Gamma, \\ \partial_t u_2 - \mu\Delta u_2 = 0 \ \text{ en } \Omega_2, \\ \dfrac{\partial u_2}{\partial \nu_2} + k u_2 = g\chi_C \ \text{ sobre } \partial\Omega_2 \setminus \Gamma, \quad \dfrac{\partial u_2}{\partial \nu_2} + k(u_2 - u_1) = 0 \ \text{ sobre } \Gamma, \end{cases}$$

donde k, g y μ son constantes positivas y ν_1, ν_2 denotan las normales unitarias exteriores a Ω_1 y Ω_2, respectivamente. Observemos que $\nu_1 = -\nu_2$ sobre Γ. La formulación variacional de este problema viene dada por

$$\begin{cases} u_1 \in L^2(0,\infty; H^1(\Omega_1)), \ u_2 \in L^2(0,\infty; H^1(\Omega_2)) \\ \dfrac{d}{dt}\displaystyle\int_{\Omega_1} u_1 v_1 \, dx + \frac{d}{dt}\int_{\Omega_2} u_2 v_2 \, dx + \mu\int_{\Omega_1} \nabla u_1 \cdot \nabla v_1 \, dx + \mu\int_{\Omega_2} \nabla u_2 \cdot \nabla v_2 \, dx \\ +k\displaystyle\int_{\partial\Omega_1\setminus\Gamma} u_1 v_1 \, ds(x) + k\int_{\partial\Omega_2\setminus\Gamma} u_2 v_2 \, ds(x) + k\int_{\Gamma} (u_1 - u_2)(v_1 - v_2) \, ds(x) = g\int_C v_2 \, dx \\ \text{en } (0,\infty), \\ \forall v_1 \in H^1(\Omega_1), \ v_2 \in H^1(\Omega_2). \end{cases}$$

En nuestro caso, hemos escogido k igual para todas las fronteras, aunque podría modificarse fácilmente. Asimismo, hemos aprovechado la simetría de los dominios, de forma que tomando

$$\hat{u_1}(x_1, x_2, t) = u_1(-x_1, x_2, t)$$

El problema queda

$$\begin{cases} \hat{u}_1, u_2 \in L^2(0,\infty; H^1(\Omega_2)), \\ \dfrac{d}{dt}\displaystyle\int_{\Omega_2} (\hat{u}_1 v_1 + u_2 v_2) \, dx + \mu\int_{\Omega_2} (\nabla \hat{u}_1 \cdot \nabla v_1 + \nabla u_2 \cdot \nabla v_2) \, dx \\ +k\displaystyle\int_{\partial\Omega_2\setminus\Gamma} (\hat{u}_1 v_1 + u_2 v_2) \, ds(x) + k\int_{\Gamma} (u_1 - \hat{u_2})(v_1 - v_2) \, ds(x) = g\int_C v_2 \, dx \\ \text{en } (0,\infty), \\ \forall v_1, v_2 \in H^1(\Omega_2). \end{cases}$$

También hemos resuelto el problema estacionario correspondiente y hemos comprobado visualmente que las soluciones del problema evolutivo convergen a las del estacionario cuando el tiempo crece.

A continuación, mostramos el código desarrollado. Como en los problemas anteriores, comenzamos definiendo las constantes, el dominio, la triangulación y el espacio sobre el que vamos a trabajar.

```
k=2 #coeficiente de robin
```

```
mu=1 #coeficiente de difusion

#figura
domain=Rectangle(Point(0,0),Point(1,1))
#triangulacion
mesh=generate_mesh(domain,50)
#espacio
V = VectorFunctionSpace(mesh, 'P', 1)
```

Indicamos los conjuntos Ω_1 y Ω_2, las cuales están sujetas a condiciones de Robin.

```
#condicion de contorno
tol= 1E-14
class Robin1(SubDomain):
    def inside(self,x,on_boundary):
        return on_boundary and x[0]<tol
class Robin2(SubDomain):
    def inside(self,x,on_boundary):
        return on_boundary and x[0]>tol
```

Creamos la frontera y los conjuntos invocando al constructor de las clases definidas arribas. Después numeramos por 1 la parte correspondiente a Ω_1 y por 2 la correspondiente a Ω_2:

```
#condicion de contorno
tol= 1E-14
class Robin1(SubDomain):
    def inside(self,x,on_boundary):
        return on_boundary and x[0]<tol
class Robin2(SubDomain):
    def inside(self,x,on_boundary):
        return on_boundary and x[0]>tol
```

Definimos la medida *ds* para poder realizar integrales de línea:

```
ds=Measure('ds',domain=mesh,subdomain_data=boundaries)
```

Indicamos la formulación variacional y resolvemos el **problema estacionario** como en los casos anteriores. No obstante, en esta ocasión debemos tener en cuenta que f es una función definida a trozos, por lo que utilizamos el operador ␣**?**␣ precedido por la condición para indicar seguidamente lo que valdrá la función en caso de verificarse la condición y, tras el operador **:**, lo que valdrá en caso contrario.

```
#formulacion variacional
u=TrialFunction(V)
v=TestFunction(V)
a=mu*inner(nabla_grad(u),nabla_grad(v))*dx+k*dot(u,v)*ds(2)
   +k*(u[0]-u[1])*(v[0]-v[1])*ds(1)
f=Expression((0,'x[0]>0.4␣and␣x[0]<0.6␣and␣x[1]>0.4␣and␣x
   [1]<0.6␣?␣50:␣0'),
```

```
                        degree = 2)
f=project(f,V)
L=dot(f,v)*dx
u=Function(V)

#resolvemos
solve(a==L,u)
```

Por último, representamos la solución obtenida del siguiente modo:

```
fig=plt.figure()
ax=fig.add_subplot(projection='3d')

def dibuja_sol(u,ax):
    u1,u2=u.split(deepcopy=True)
    x=np.linspace(-1,1,200)
    y=np.linspace(0,1,100)
    def v(x,y):
        if x<0:
            v=u1(-x,y)
        else:
            v=u2(x,y)
        return v
    X,Y=np.meshgrid(x,y)
    Z=np.zeros((len(y),len(x)))
    for i in range(0,len(y)):
        for j in range(0,len(x)):
            Z[i,j]=v(X[i,j],Y[i,j])
    ax.plot_surface(X,Y,Z,cmap='gnuplot')
    ax.set_xlim(-1,1)
    ax.set_ylim(0,2)
    ax.set_zlim(0,2)
    plt.show()
```

Mostramos el gráfico obtenido. Cabe destacar que se observa un desnivel entre las dos hipotéticas habitaciones que representa la pared que las separa.

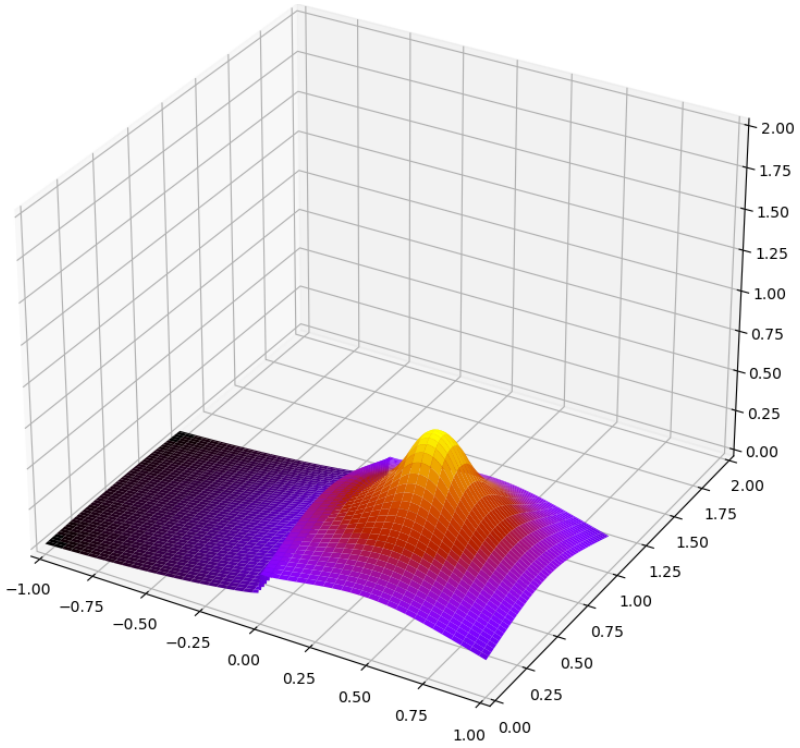

Figura 6.7: Representación de la solución del problema del calor estacionario.

Asimismo, mostramos la perspectiva en planta de dicha solución para, además de tener información sobre la temperatura alcanzada en las distintas zonas, poder observar la difusión con mayor facilidad.

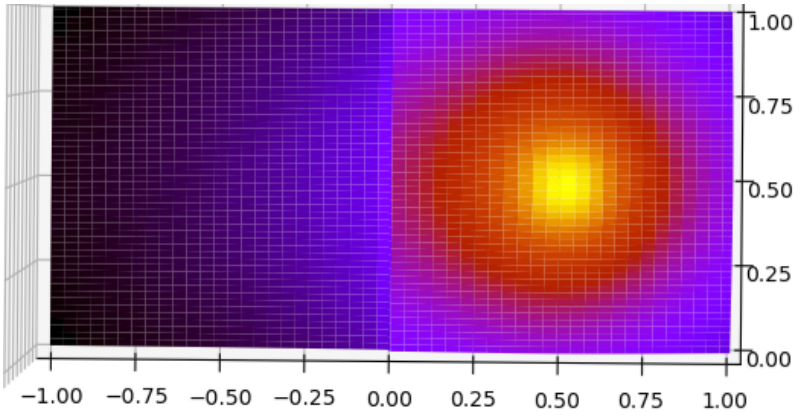

Figura 6.8: Visualización en planta de la solución del problema del calor estacionario.

Efectivamente, se observa cómo, desde el punto de vista de la aplicación, la fuente de calor va difundiéndose de forma radial alcanzando la habitación contigua con menor intensidad debido a la presencia de la pared.

A continuación, vamos a resolver el **problema evolutivo**. Antes de definir la nueva formulación variacional introduciendo el paso del tiempo y los términos u_n, es

necesario comentar la anterior, ya que el software produce error si hay más de una forma bilineal definida durante la misma ejecución.

```
h=1/250
num_steps=220
u_n=Constant((0,0))
u_n=project(u_n,V)
#formulacion variacional
a=dot(u,v)*dx+h*mu*inner(nabla_grad(u),nabla_grad(v))*dx+h*
   k*dot(u,v)*ds(2)+h*k*(u[0]-u[1])*(v[0]-v[1])*ds(1)
L=dot(u_n,v)*dx+h*dot(f,v)*dx
u=Function(V)
```

Finalmente, resolvemos de forma iterada. En cada paso calculamos la solución u y representamos y actualizamos u_n con el valor obtenido. Las visualizaciones se van almacenando de forma simultánea al bucle dando lugar a una animación al final del método.

```
def anima_difusion(num_steps):
    fig=plt.figure()
    ax=fig.add_subplot(projection='3d')
    def actualizar(i):
        #Resolucion numerica
        solve(a==L,u)
        #Almacenamos representacion
        dibuja_sol(u_n,ax)
        #Actualizacion de la solucion
        u_n.assign(u)
        print(i)
    writervideo=animation.FFMpegWriter(fps=20)
    anim=animation.FuncAnimation(fig,actualizar,range(0,
       num_steps),interval=1,repeat=False,blit=False)
    anim.save('difusion.mp4',writer=writervideo)
    return anim
```

Terminamos ejecutando el método y mostrando algunos instante de la animación obtenida:

```
anima_difusion(num_steps)
```

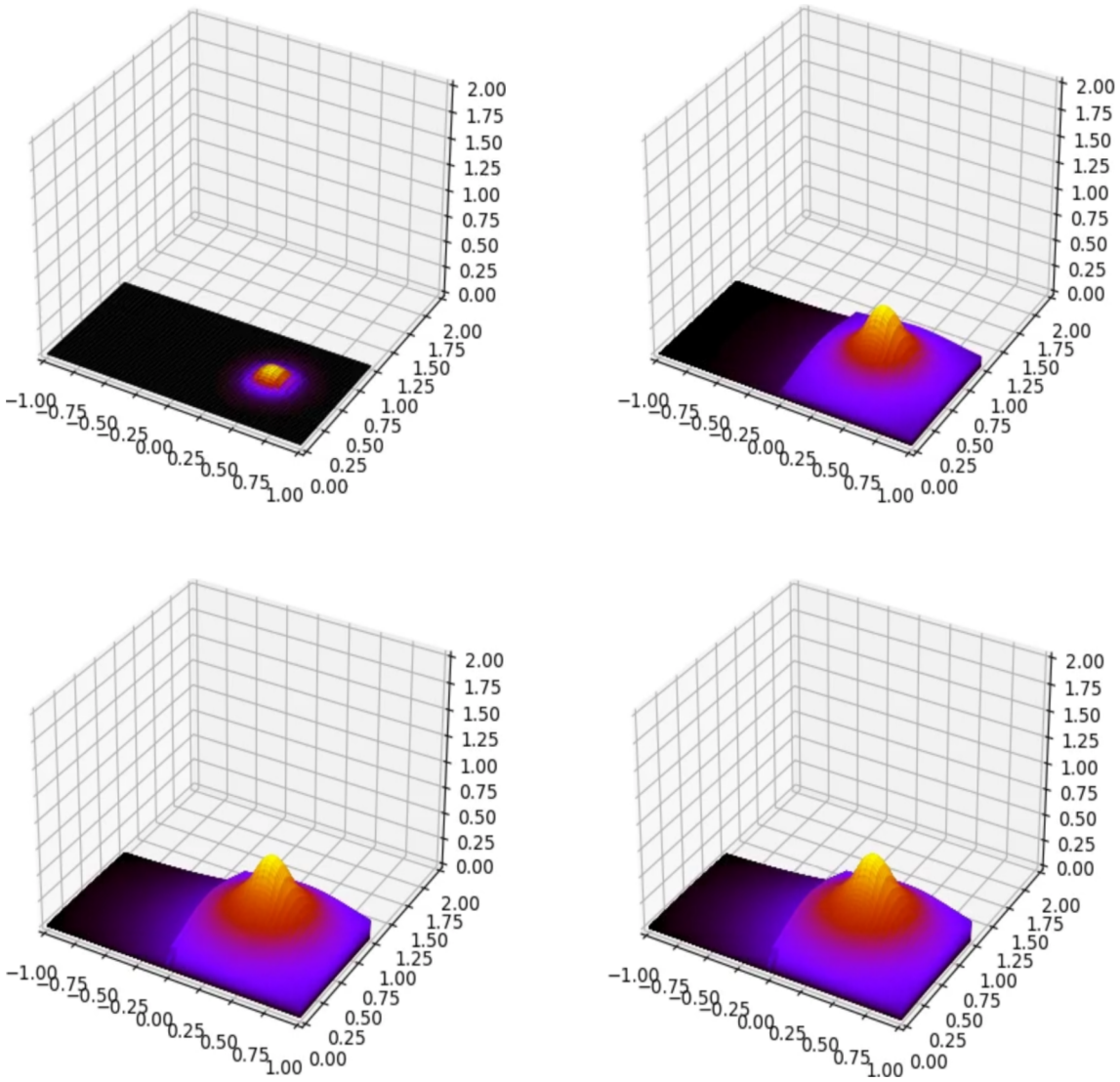

Figura 6.9: Fotogramas pertenecientes a la animación generada para el problema de difusión del calor evolutivo.

7. Bibliografía

[1] M. Crouzeix, A.L. Mignot, *Analyse numérique des équations différentielles*, Masson, Paris, 1984.

[2] T. Sauer, *Numerical Analysis*, Pearson, Hoboken (New Jersey), 2018.

[3] W. H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, 3rd edition, 2007.

[4] C.A. Kennedy, M.H. Carpenter, *Diagonally Implicit Runge-Kutta Methods for Ordinary Differential Equations. A Review*, National Aeronautics and Space Administration Langley Research Center Hampton, Virginia, 2016.

[5] P.A. Raviart, J.M. Thomas, *Introduction à l'analyse numérique des équations aux dérivées partielles*, Masson, Paris, 1988.

[6] D. Gilbarg, N.S. Trudinger, *Elliptic partial differential equations of second order*, Springer, Berlin, 1998.

[7] P. Grisvard, *Elliptic Problems in Nonsmooth Domains*, Pitman, Boston, 1985.

[8] H.P. Langtangen, A. Logg, *Solving PDEs in Python. The FeniCs Tutorial I*, Springer, Open Access, 2017.

[9] Documentación oficial de numpy: https://numpy.org/doc/stable/

[10] Documentación oficial de matplotlib: https://matplotlib.org/stable/

[11] Documentación oficial de scipy: https://docs.scipy.org/doc/scipy/reference/index.html#scipy-api

[12] Documentación oficial de animation: https://matplotlib.org/stable/api/animation_api.html

[13] Guía para configurar FFmpeg en Windows: https://es.101-help.com/f9ac8627f0-como-instalar-y-usar-ffmpeg-en-windows-10/

[14] Guía de descarga de FFmpeg: https://browzwear.com/services/downloads/ffmpeg/#:~:text=Browzwear%20Help%20Center.-,Windows,files%2C%20click%20the%20Windows%20icon

[15] Tutorial sobre creación de animaciones con Python: https://www.youtube.com/watch?v=YP1zBhMndI0

[16] Entrada de Stack Overflow sobre uso de leyendas en subplots de matplotlib: https://stackoverflow.com/questions/27016904/matplotlib-legends-in-subplot

[17] Repositorio de Github oficial de FEniCs. https://github.com/FEniCS/dolfinx#ubuntu-packages

[18] Documentación oficial de FEniCs. https://fenicsproject.org/

[19] Guía de instalación de Python en Ubuntu: https://www.ochobitshacenunbyte.com/2022/12/01/como-instalar-la-ultima-version-de-python-en-ubuntu/

[20] Página oficial de VirtualBox: https://www.virtualbox.org/

[21] Guía para actualizar VirtualBox: https://www.solvetic.com/tutoriales/article/9565-actualizar-virtualbox-sin-perder-maquina-virtual-windows-10/

[22] Entrada de ask Ubuntu para arreglar error de VirtualBox que impide abrir la terminal de Ubuntu 22.04. https://askubuntu.com/questions/1435918/terminal-not-opening-on-ubuntu-22-04-on-virtual-box-7-0-0

[23] Instrucciones de configuración de Ubuntu: https://ubuntu.com/tutorials/install-ubuntu-desktop#1-overview

[24] Enlace de descarga del disco ISO de Ubuntu 22.04: https://releases.ubuntu.com/jammy/

[25] Website oficial de ParaView: https://www.paraview.org

[26] Website oficial de Anaconda: https://www.anaconda.com/

[27] Website oficial de Docker: https://www.docker.com/

[28] Tutorial para instalar FEniCS en Windows10 utilizando el subsistema de Linux para Windows: https://www.youtube.com/watch?v=wFh5gPv_R2c