UNIVERSIDAD DE CÁDIZ

GRADO EN MATEMÁTICAS

RESOLUCIÓN NUMÉRICA DE ECUACIONES DIFERENCIALES DIFUSAS

TRABAJO FIN DE GRADO

CURSO ACADÉMICO: 2017/2018

JOSÉ CARLOS GARCÍA ORTEGA

Dr. Rafael Rodríguez Galván

Dr. Jesús Medina Moreno

UNIVERSIDAD DE CÁDIZ

GRADO EN MATEMÁTICAS

RESOLUCIÓN NUMÉRICA DE ECUACIONES DIFERENCIALES DIFUSAS

TRABAJO FIN DE GRADO

CURSO ACADÉMICO: 2017/2018

JOSÉ CARLOS GARCÍA ORTEGA

Dr. Rafael Rodríguez Galván Dr. Jesús Medina Moreno

FIRMA DEL ALUMNO FIRMA DEL TUTOR FIRMA DEL TUTOR

Jerez de la Frontera, Cádiz, junio 2019

Abstract

This work tries to introduce fuzzy logic theory, and search new concepts related with the conception of fuzzy analysis theory with the objective of find a concept for fuzzy differential equations. The fundamental pillars on which this theory is based lie in the power of Zadeh's extension principle, fuzzy numbers, and over every different arithmetic operation defined over fuzzy numbers.

By some regularity conditions not too restrictive, you can find a relationship between solving a fuzzy differential equation and solving an ordinary differential equation, this theoretical framework, invites and accompanies build numerical methods based on classical numerical methods that are known worldwide.

This work also introduces some models that can be built taking into account the fuzzy nature of real life.

Finally, this work also involves high performance computational techniques, to achieve the best energy performance and in a matter of time. And it will be seen, that high-performance computational techniques can change the results of tests quite remarkably.

Resumen

Este trabajo trata de introducir la teoría de lógica difusa, y buscar nuevos conceptos relacionados con la concepción de una teoría de análisis difuso para finalmente llegar al concepto de ecuación diferencial difusa. Los pilares fundamentales donde se va a basar esta teoría subyacen en el poder de la definición del principio de extensión de Zadeh, los conceptos de números difusos, y las diferentes operaciones aritméticas que se pueden definir.

Mediante unas condiciones de regularidad no demasiado restrictivas, se puede encontrar una relación entre resolver una ecuación diferencial difusa y una ecuación diferencial determinista, este marco teórico, invita y acompaña a construir métodos numéricos basados en los métodos numéricos clásicos que ya son conocidos por todo el mundo.

Este trabajo, también introduce algunos modelos que se pueden construir teniendo en cuenta la naturaleza difusa de la vida real.

Finalmente, en esta memoria también se trabajan técnicas computacionales de alto rendimiento, para conseguir resolver los problemas de forma eficiente en cuestión de tiempos y de energía. Estas observaciones nos mostrará que las técnicas computacionales de alto rendimiento pueden cambiar de forma notable los resultados de las pruebas.

Agradecimientos

Este trabajo no habría sido posible sin las personas que han ido pasando por mi vida a lo largo de los años.

En primer lugar, quiero agradecer a mis padres por su cariño infinito, su apoyo, su trabajo y su paciencia, donde quiero destacar también el visionario carácter de mi madre que cuando yo tan sólo tenía 6 años vio la importancia que iba a tener los ordenadores en el futuro, y que a día de hoy, gracias a ser una visionaria sé lo que sé y ha hecho que los ordenadores y yo formemos uno. También quiero destacar el carácter luchador de mi padre, una persona súper trabajadora que nunca para, que siempre está tratando de ayudar a la gente de su alrededor sin importar lo cansado que él esté.

También quiero destacar el carácter luchador de mi hermana de quien he aprendido muchísimo.

No podía faltar aquí una especial mención a mis abuelos por su dedicación y cariño, en especial a mi abuelo Pepe que estaba cuando lo necesitaba, me cuidaba, me mimaba y curaba mis impuntualidades llevándome a la universidad en coche.

Tampoco puedo dejar de agradecer a Marta, por lo agusto que me haces sentir, por la inspiración y por mostrarme una definición tangible de lo que es tocar el infinito.

A mis padrinos por celebrar conmigo todos los logros y avances a lo largo de estos años, y ser para mi unos segundos padres.

A mis compañeros de la universidad, que han hecho más ameno, más divertidos y me han hecho crecer más como persona, me gustaría hacer especial mención a Virginia, quien para mi se ha convertido en una amiga, de quien he aprendido, me he asombrado por su capacidad luchadora y de quien también he usado sus apuntes para sacar adelante asignaturas. Del mismo modo, no puedo olvidarme tampoco de Pilar, quien ha hecho más amenas las clases, hemos colaborado en trabajos juntos y hemos compartido apuntes y conocimientos, que a día de hoy también considero mi amiga.

A mi vecino Paco por meterme el gusanillo de la informática, que a día de hoy sigue más vivo que nunca.

Finalmente, quiero agradecer a mis tutores Rafa y Jesús por su apoyo, sus correcciones y consejos, los cuales han ayudado que este trabajo llegue al nivel que tiene ahora. Quiero hacer especial mención a Rafa, que no sólo ha sido mi tutor del TFG sino que también estuve con él durante 2 años de alumno colaborador, donde nacieron interesantes discusiones, aprendí mucho y me ayudó a abrirme al mundo del software libre.

ÍNDICE GENERAL

NOTACIÓN

$\mathbb{U}$	$\triangleq$	Conjunto universo
$cl\, A$	$\triangleq$	Clausura de A
$\mathcal{F}(\mathbb{U})$	$\triangleq$	La familia de conjuntos difusos de $\mathbb{U}$.
$\mathcal{F}_{\mathcal{H}}(\mathbb{U})$	$\triangleq$	La familia de conjuntos difusos de $\mathbb{U}$ tales que sus α-cortes son no vacios, compactos en $\mathbb{U}$
$\mathcal{F}_{\mathcal{C}}(\mathbb{U})$	$\triangleq$	La familia de conjuntos difusos de $\mathbb{U}$ tales que sus α-cortes son no vacios, compactos y convexos en $\mathbb{U}$
$\ominus_H$	$\triangleq$	Diferencia de Hukuhara
$(a; b; c)$	$\triangleq$	Número triangular

MEDIOS INFORMÁTICOS

"Mi trabajo en el software libre está motivado por un objetivo idealista: difundir libertad y cooperación. Quiero motivar la expansión del software libre, reemplazando el software privativo que prohibe la cooperación, y de este modo hacer nuestra sociedad mejor."

– Richard Stallman

Declaración sobre las licencias y ámbito científico

Todo el código creado para generar este proyecto va a estar disponible para descargar, modificar y compartir sin ningún tipo de limitación. Te animo a hacer tuyo cualquier código que necesites para tus proyectos, los modifiques según tus necesidades y que luego compartas el código para que otras personas puedan seguir mejorándolo.

Especificaciones

Núcleo	Linux debian 4.9.0-8-amd64
SO	Debian 9.5 Stretch (Stable)
GPU	Nvidia 930M
CPU	Intel i7-5500U
RAM	12GB

Código fuente del trabajo

Cumpliendo la licencia que se expone en la primera página del TFG, se puede descargar una copia completa de este trabajo, junto a todo el código necesario para generarlo a través del

siguiente enlace: https://github.com/JoseCarlosGarcia95/TFG2018

1 PREFACIO

1.1. Marco teórico

En este apartado se hace un repaso somero a todos los conceptos necesarios para introducir los objetivos del trabajo.
En primer lugar, lo primero que se va a necesitar para este trabajo es el concepto de conjunto difuso, que nace de la naturaleza del lenguaje humano, donde nuestra mente se siente más cómoda con términos cualitativos que con términos cuantitativos.

De esta característica del lenguaje, en 1965, Lofti A. Zadeh publicó lo que hoy se conoce como teoría de conjuntos difusos. La teoría de conjuntos difusos generaliza la idea detrás de la función característica de los conjuntos clásicos, que se define de la siguiente forma:

Sea A un conjunto cualesquiera, se define la función característica $\chi_A(x)$ de A como:

$$\chi_A(x) = \begin{cases} 1 & si \quad x \in A \\ 0 & si \quad x \notin A \end{cases} .$$

De esta manera, para definir lo que es un conjunto difuso se generaliza la idea de función característica a una función que toma valores en $[0, 1]$, de forma que un conjunto difuso B se define como un par ordenado dado por $(\mathbb{U}, \mu_B)$ donde $\mu_B : \mathbb{U} \longrightarrow [0, 1]$ y $\mathbb{U}$ es el conjunto de discurso. (Ver definición subconjunto difuso)

Dentro de la teoría de conjuntos difusos también encontramos lo que llamamos número difuso, que es un conjunto difuso que cumple que es un conjunto normal y convexo. Un ejemplo clásico de número difuso corresponde a los números triangulares, se puede representar de la siguiente forma:

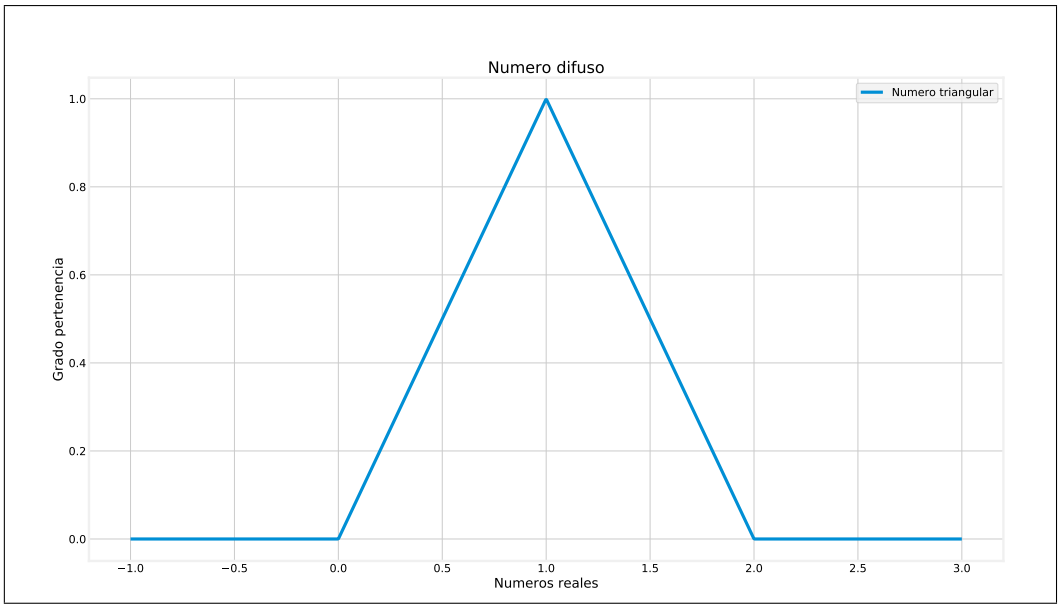


Figura 1.1: Ejemplo de número triangular

Por otro lado, como ocurren en los números reales, también se le pueden asociar operaciones aritméticas básicas como la suma, el producto, la resta y la división. Las definiciones de suma, producto y división de números difusos funciona de forma bastante intuitiva, sin embargo, definir el concepto de resta de dos números difusos es más complejo, concepto que nos permitirá introducir derivadas de subconjuntos difusos, que es uno de los objetivos.

Otro concepto interesante a tener en cuenta y que marcó una revolución es la introducción del principio de extensión de Zadeh [1], que permite calcular la imagen de un conjunto difuso mediante una función real. Todos estos conceptos introducidos por Lotfi A. Zadeh dan lugar a un marco teórico que permite desarrollar conceptos clásicos del análisis matemático en un contexto difuso.

1.2. Objetivos de este trabajo

Este trabajo trata de generalizar el concepto de ecuación diferencial a un marco difuso utilizando la teoría de conjuntos difusos. Para resolver las ecuaciones planetadas, se van a utilizar métodos numéricos y técnicas computacionales avanzadas, se puede decir que los objetivos principales de este trabajo son:

- Conocer el concepto de número difuso.
- Introducir teoremas clásicos del análisis mediante la teoría de conjuntos difusos.
- Desarrollar el concepto de ecuación diferencial difusa.

- Hacer un análisis bibliográfico de los conceptos de análisis difuso.
- Comparar las soluciones de una ecuación diferencial difusa y una ecuación diferencial ordinaria.
- Resolver numéricamente ecuaciones diferenciales difusas.
- Introducir técnicas informáticas avanzadas para mejorar el rendimiento de los algoritmos numéricos desarrollados.
- Trabajar tecnologías pensando en el medioambiente. Un software que consume menos energía es mejor para el medioambiente.

2 CONJUNTOS DIFUSOS

"A medida que aumenta la complejidad, las declaraciones precisas pierden significado y las declaraciones significativas pierden precisión"

– Lotfi A. Zadeh

El concepto de conjunto difuso fue introducido en 1965 por Lotfi A. Zadeh, y desde su introducción se ha estudiado profundamente, y se ha considerado para extender otras teorías, permitiendo entre otras cosas el tratamiento de bases de datos con incertidumbre.

Su desarrollo fue motivado por la naturaleza cualitativa que se encuentra en el lenguaje coloquial. Los intentos de usar tecnología computacional para procesar modelos usando la forma probabilística de la incertidumbre, se ha demostrado que no han sido del todo satisfactorios.

Se puede decir que la probabilidad calcula de forma cuantitativa de que se verifique un proceso bien definido (por ejemplo, las caras de una moneda), sin embargo, la teoría difusa viene a analizar la incertidumbre que existe en elementos bien conocidos. ¿Es este color violeta o es más bien azul? ¿Está la temperatura alta o baja? Este último tipo de modelos son esenciales para solucionar problemas técnicos (teoría de control), economía (análisis de mercado) y otros problemas influenciados por la naturaleza cualitativa del lenguaje humano [2].

El contenido de este capítulo se basa en las definiciones que aparecen en la referencia [1].

2.1. Subconjuntos difusos

La teoría clásica de conjuntos solo abarca la posibilidad de que un elemento pertenezca o no, a un conjunto. Pero la realidad no es así, y pueden existir ciertos casos en lo que la pertenencia o no, a un conjunto haya que definirla mediante un grado de pertenencia.

2.1.1. Subconjuntos difusos

En primer lugar, se introduce el concepto de conjunto difuso que servirá para formalizar el concepto de conjunto con grados de pertenencia.

Definición 1 (Subconjunto difuso)**.** *Un subconjunto difuso A es un par ordenado $(\mathbb{U}, \mu_A)$ con:*

$$\mu_A : \mathbb{U} \longrightarrow [0, 1].$$

Se denomina a μ_A función de pertenencia.

Denotamos por $\mathcal{F}(\mathbb{U})$ al conjunto de todos los conjuntos difusos de $\mathbb{U}$.
Esta función de pertenencia no define necesariamente una probabilidad, y no hace referencia a la probabilidad de que una persona sea alta o baja, si no que da una medida de cuan alta o baja es.

Por tanto, es natural definir la igualdad de conjuntos de la siguiente forma:

Definición 2 (Igualdad de conjuntos difusos)**.** *Se dice que dos conjuntos difusos A y B en $\mathbb{U}$ son iguales si para todo $x \in \mathbb{U}$, se cumple $\mu_A(x) = \mu_B(x)$.*

Si se cumpliese que $\mu_A(\mathbb{U}) = \{0, 1\}$ se obtiene que A es un conjunto clásico, y μ_A es la función característica de A. En este caso, si $x \in A$, entonces $\mu_A(x) = 1$, y por el contrario si $x \notin A$ entonces $\mu_A(x) = 0$.
Por tanto, la función μ_A representa una generalización del concepto de función característica clásica donde μ_A representa el grado de pertenencia a un conjunto.

2.1.2. α-corte

Ahora se introduce un concepto fundamental en la teoría de conjuntos difusos es el concepto de α-corte, que permitirá crear particiones de los conjuntos separados por los valores de la función de pertenencia.

Definición 3 (α-corte)**.** *Dado un conjunto difuso A, los α-corte son los subconjuntos clásicos dados por:*

$$[A]_\alpha = \begin{cases} \{x \in \mathbb{U} : \mu_A(x) \geq \alpha\} & si \quad \alpha \in (0, 1] \\ cl\{x \in \mathbb{U} : \mu_A(x) > 0\} & si \quad \alpha = 0 \end{cases},$$

donde cl define la clausura topológica del conjunto en el intervalo unidad. Además, se define:

$$soporte\ A = \{x \in \mathbb{U} : \mu_A(x) > 0\},$$

$$núcleo\ A = \{x \in \mathbb{U} : \mu_A(x) = 1\}.$$

Denotamos por $\mathcal{F}_{\mathcal{H}}(\mathbb{U})$ a la familia de conjuntos de $\mathbb{U}$ tales que sus α-cortes son no vacíos y compactos.
De esta definición se puede extraer que un conjunto difuso también puede estar definido por sus α-cortes, de manera que dos conjuntos difusos A y B son iguales, si todos sus α-cortes son iguales.

A partir de los α-cortes se puede construir una función de pertenencia de la siguiente forma [3]

$$\mu_A(x) = \max\{\alpha A_\alpha(x) : \alpha \in [0,1]\},$$

$$\text{donde } A_\alpha(x) = \begin{cases} 1 & si \quad x \in [A]_\alpha \\ 0 & si \quad x \notin [A]_\alpha \end{cases}.$$

Desde aquí, este estudio se centrará en los α-cortes de los conjuntos difusos.

2.2. Números difusos

Para poder trabajar con sistemas de ecuaciones diferenciales difusas, es necesario introducir el concepto de número difuso.
Se da en primer lugar dos definiciones necesarias para definir el concepto de número difuso.

Definición 4 (Conjunto difuso normal)**.** *Un conjunto difuso A es normal si núcleo $A \neq \emptyset$.*

Definición 5 (Conjunto difuso convexo)**.** *Un conjunto difuso A es convexo si su función de pertenencia es cuasicóncava, esto es:*

$$\mu_A(\lambda x + (1-\lambda)y) \geq \min\{\mu_A(x), \mu_A(y)\}, \textit{ para todo } \lambda \in [0,1], x, y \in \mathbb{U}.$$

En este caso, si $\mathbb{U} = \mathbb{R}$ se obtiene que si A es un conjunto difuso convexo, entonces los α-cortes son intervalos. Se denota de la siguiente forma:

$$[X_0]_\alpha = [(x_0)_\alpha^-, (x_0)_\alpha^+].$$

Finalmente, se define el concepto de número difuso:

Definición 6 (Número difuso)**.** *Un conjunto difuso A es un número difuso si $\mathbb{U} = \mathbb{R}$, A es normal y convexo, y además su función de pertenencia es continua por la derecha.*

Se observa que si $x \in \mathbb{R}$, el conjunto $\{x\}$ puede representar un número difuso, donde su función de pertenencia es su función característica.
Denotamos por $\mathcal{F}_\mathcal{C}(\mathbb{U})$ la familia de conjuntos de $\mathbb{U}$ tales que sus α-cortes no vacíos, compactos y convexos en $\mathbb{U}$.

2.2.1. Caracterización números difusos

A continuación, se introducen dos teoremas que permitirán caracterizar los números difusos. Las demostraciones de estos dos resultados se pueden encontrar en [1].

Teorema 1 (Teorema de apilamiento)**.** *Sea A un número difuso, entonces:*

1. *Sus α-cortes son intervalos cerrados no vacíos para todo $\alpha \in [0,1]$.*

2. *Si* $0 \leq \alpha_1 \leq \alpha_2 \leq 1$ *entonces* $[A]_{\alpha_1}. \subset [A]_{\alpha_2}$

3. *Para toda sucesión no decreciente de* $\alpha_n \in [0,1]$ *tal que* a_n *tiende a* α *se obtiene que:*

$$\bigcap_{n=1}^{\infty} [A]_{\alpha_n} = [A]_{\alpha}.$$

4. *Para toda sucesión no creciente* $\alpha_n \in [0,1]$ *convergente a* 0 *se obtiene:*

$$cl\left(\bigcup_{n=1}^{\infty} [A]_{\alpha_n}\right) = [A]_0.$$

Teorema 2 (Teorema de caracterización)**.** *Sea* $A = \{A_\alpha : \alpha \in [0,1]\}$ *una familia de subconjuntos de* $\mathbb{R}$ *tal que:*

1. *Sus* A_α *son intervalos cerrados no vacíos para todo* $\alpha \in [0,1]$.

2. *Si* $0 \leq \alpha_1 \leq \alpha_2 \leq 1$ *entonces* $A_{\alpha_1} \subset A_{\alpha_2}$.

3. *Para toda sucesión no decreciente* $\alpha_n \in [0,1]$ *tal que* α_n *tiende a* α *se obtiene que*

$$\bigcap_{n=1}^{\infty} A_{\alpha_n} = A_{\alpha}.$$

4. *Para toda sucesión no creciente* $\alpha_n \in [0,1]$ *convergente a* 0 *se obtiene:*

$$cl\left(\bigcup_{n=1}^{\infty} A_{\alpha_n}\right) = A_0.$$

Entonces, A *es un número difuso.*

Ejemplo 1. *Sea* $\mathbb{U} = \mathbb{R}$ *y sea* $\mu_{\mathcal{A}} : \mathbb{R} \longrightarrow [0,1]$ *definida de la siguiente forma:*

$$\mu_{\mathcal{A}}(x) = \begin{cases} \frac{x-a}{b-a} & si \quad x \in [a,b] \\ \frac{c-x}{c-b} & si \quad x \in (b,c] \\ 0, & si \quad x \notin [a,c] \end{cases}.$$

Con $a < b < c$. *Entonces, un número difuso definido de la forma anterior es triangular se denota (a; b; c), y se puede representar como en la figura 2.1*

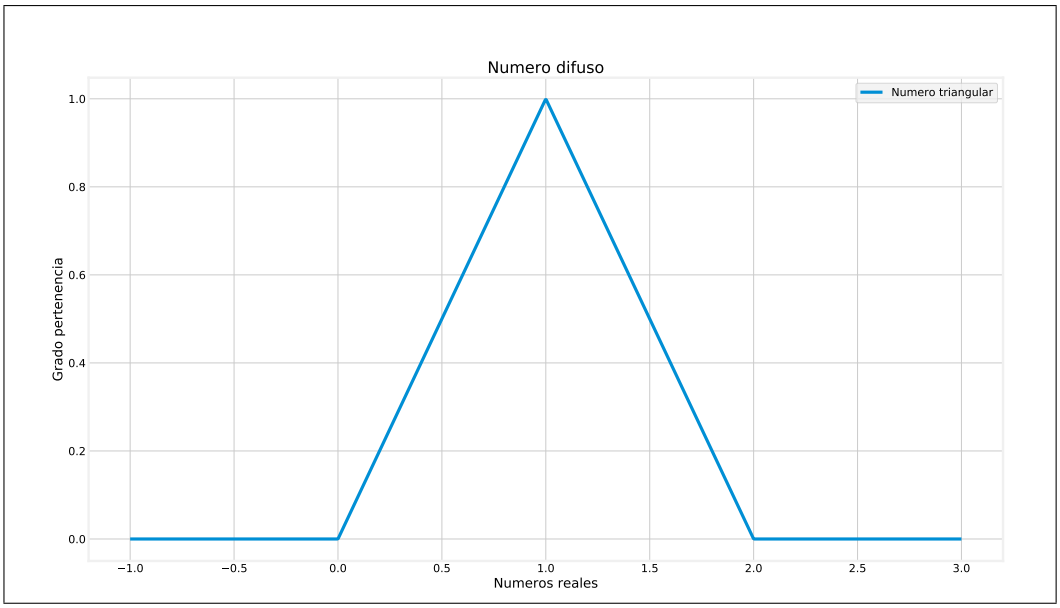


Figura 2.1: Ejemplo de número triangular

2.3. Principio de extensión de Zadeh

Estaría bien que dado un conjunto difuso, y una función clásica entre dos universos, se pudiera calcular el conjunto difuso imagen del conjunto difuso a través de la función clásica, para esto se introduce el principio de extensión de Zadeh.

Definición 7 (Principio de extensión de Zadeh)**.** *Sean $\mathbb{U}$ y $\mathbb{V}$ dos universos, y sea $f : \mathbb{U} \longrightarrow \mathbb{V}$ una función clásica. Se define el principio de extensión de Zadeh, para todo conjunto difuso (A, μ_A) como el conjunto difuso $(\hat{f}(A), \mu_{\hat{f}(A)})$ donde*

$$\mu_{\hat{f}(A)}(y) = \begin{cases} \sup_{x \in f^{-1}(y)} \mu_A(x) & si \quad f^{-1}(y) \neq \emptyset \\ 0 & si \quad f^{-1} = \emptyset \end{cases} .$$

Se puede observar, que si la función es inyectiva, la función de pertenencia se simplificaría de la siguiente forma

$$\mu_{\hat{f}(A)}(y) = \begin{cases} \mu_A(f^{-1}(y)) & si \quad f^{-1}(y) \neq \emptyset \\ 0 & si \quad f^{-1} = \emptyset \end{cases} .$$

2.3.1. Teoremas de continuidad

Sería ideal que no importase el orden con el que se obtienen los α-cortes de la imagen de una función mediante el principio de extensión de Zadeh, y esto lo asegura el siguiente teorema.

Teorema 3. *Sea $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ una función.*

1. *Si f es sobreyectiva, entonces $[\hat{f}(A)]_\alpha = f([A]_\alpha)$ si y solo si $\sup\{\mu_A(x) : x \in f^{-1}(y)\}$ es alcanzable para todo $y \in \mathbb{R}^m$*
2. *Si f es continua, entonces $\hat{f} : \mathcal{F}_\mathcal{H}(\mathbb{R}^n) \longrightarrow \mathcal{F}_\mathcal{H}(\mathbb{R}^n)$ está bien definido y además,*

$$[\hat{f}(A)]_\alpha = f([A]_\alpha),$$

para todo $\alpha \in [0, 1]$

La primera implicación es clara por la definición de principio de extensión de Zadeh, para la segunda implicación se introduce un teorema más general:

Teorema 4. *Sean $\mathbb{U}$ y $\mathbb{V}$ unos espacios de Haussdorf, y sea $f : \mathbb{U} \longrightarrow \mathbb{V}$ una función. Si f es continua, entonces $\hat{f} : \mathcal{F}_\mathcal{H}(\mathbb{R}^n) \longrightarrow \mathcal{F}_\mathcal{H}(\mathbb{R}^n)$ está bien definida y*

$$[\hat{f}(A)]_\alpha = f([A]_\alpha),$$

para todo $\alpha \in [0, 1]$.

Demostración. Por la definición del principio de Zadeh, se obtiene que $\hat{f}(A)$ es un subconjunto difuso de $\mathbb{V}$.
Para probar que $\hat{f} : \mathcal{F}_\mathcal{H}(\mathbb{R}^n) \longrightarrow \mathcal{F}_\mathcal{H}(\mathbb{R}^n)$ hay que ver que todos los α-cortes $[\hat{f}(A)]_\alpha$ son no vacíos y compactos en $\mathbb{V}$.
Sabemos que f es continua por hipótesis, la imagen de compactos, son compactos, por tanto, solo hay que probar $[\hat{f}(A)]_\alpha = f([A]_\alpha)$.
Se procede por doble inclusión.

- $[\hat{f}(A)]_\alpha \supseteq f([A]_\alpha)$. Sea $y \in f([A]_\alpha)$. Por tanto, existe al menos un $x \in [A]_\alpha$ tal que $f(x) = y$. Por el principio de extensión de Zadeh, obtenemos

$$\mu_{\hat{f}(A)}(y) = \sup_{x \in f^{-1}(y)} \mu_A(x) \geq \alpha.$$

  De donde, $y \in [\hat{f}(A)]_\alpha$

- Por otro lado, hay que ver que $[\hat{f}(A)]_\alpha \subseteq f([A]_\alpha)$. Dado que $\mathbb{V}$ y $\mathbb{U}$ son espacios de Haussdorf, un punto $y \in \mathbb{V}$ es cerrado. Y además, dado que f es continua, $f^{-1}(y)$ es cerrado. Dado que $[A]_0$ es compacto, ya que es la clausura, la intersección de compactos también es compacto, por tanto $f^{-1}(y) \cap [A]_0$ es compacto. Para $\alpha > 0$, sea $y \in [\hat{f}(A)]_\alpha$. Entonces $\mu_{\hat{f}(A)}(y) = \sup_{x \in f^{-1}(y)} \mu_A(x) \geq \alpha > 0$, por tanto, existe un $x \in f^{-1}(y)$ y en consecuencia $f^{-1}(y) \cap [A]_0 \neq \emptyset$.

  Finalmente, debido a que $\mu_A(x)$ es continúa por la derecha, y $f^{-1}(y) \cap [A]_0$ es compacto, existe un $x \in f^{-1}(y) \cap [A]_0$ con $\mu_{\hat{f}(A)}(y) = \mu_A(x) \geq \alpha$. Esto es porque $y = f(x)$ para algún $x \in [A]_\alpha$.

  Para $\alpha = 0$ se obtiene:

$$\bigcup_{\alpha\in(0,1]} [\hat{f}(A)]_\alpha = \bigcup_{\alpha\in(0,1]} f([A]_\alpha) \subset f([A]_0).$$

Dado que $f([\mathcal{A}]_0)$ es cerrado:

$$[\hat{f}(\mathcal{A})]_0 = cl\left(\bigcup_{\alpha\in(0,1]} [\hat{f}(\mathcal{A})]_\alpha\right) = cl\left(\bigcup_{\alpha\in(0,1]} f([\mathcal{A}]_\alpha)\right) \subset f([\mathcal{A}]_0).$$

Y por la doble inclusión anterior, se tiene que $[\hat{f}(\mathcal{A})]_\alpha = f([\mathcal{A}]_\alpha)$ para todo $\alpha \in [0, 1]$ □

2.4. Aritmética difusa

El siguiente paso para poder construir métodos numéricos es definir las operaciones aritméticas básicas entre conjuntos difusos. La definiciones de estas operaciones son bastante naturales, pero pueden ocasionar algunos problemas en ciertos escenarios.
Debido a la equivalencia entre trabajar con conjuntos difusos, y sus α-cortes, nos centraremos en dar todas las operaciones en términos de α-cortes.
En primer lugar, se recordarán las definiciones habituales de las operaciones aritmética para la teoría de conjuntos clásicos.

2.4.1. Aritmética en conjuntos clásicos

Sean A, B dos conjuntos entonces:

- $A + B = \{a + b : a \in A, b \in B\}$.
- $A - B = \{a - b : a \in A, b \in B\}$.
- $A * B = \{ab : a \in A, b \in B\}$.
- $A/B = \{a/b : a \in A, b \in B\}$.

Una vez recordadas las operaciones aritmética en conjuntos clásicos, se van a generalizar para conjuntos difusos.

2.4.2. Aritmética en conjuntos difusos

Sean μ_A, μ_B dos funciones de pertenencia y sea $\odot \in \{+, -, \cdot, \div\}$ se define la función de pertenencia de la operación aritmética como:

$$\mu_{A\odot B}(c) = \sup_{a\odot b=c} \min\{\mu_A(a), \mu_B(b)\}.$$

Y dado que las operaciones aritméticas son funciones continuas, es equivalente trabajar con los α-cortes, aplicando el principio de extensión de Zadeh tenemos:
Sean A y B dos números difusos no interactivos (Ver definición 13) con $\alpha-$cortes dados por $[A]_\alpha = [a_\alpha^-, a_\alpha^+]$ y $[B]_\alpha = [b_\alpha^-, b_\alpha^+]$, se puede definir entonces las operaciones aritméticas como [1]:

$$[A+B]_\alpha = [a_\alpha^- + b_\alpha^-, a_\alpha^+ + b_\alpha^+].$$

$$[A-B]_\alpha = [a_\alpha^- - b_\alpha^+, a_\alpha^+ - b_\alpha^-].$$

$$[A \cdot B]_\alpha = \left[\min_{s,r\in\{-,+\}} a_\alpha^s \cdot b_\alpha^r, \max_{s,r\in\{-,+\}} a_\alpha^s \cdot b_\alpha^r \right].$$

$$[A \div B]_\alpha = \left[\min_{s,r\in\{-,+\}} \frac{a_\alpha^s}{b_\alpha^r}, \max_{s,r\in\{-,+\}} \frac{a_\alpha^s}{b_\alpha^r} \right].$$

Problemas al definir estas operaciones aritméticas

El objetivo final es definir la diferencial de una función. Para funciones escalares de una sola variable se define la diferencial como:

$$\lim_{h\to 0} \frac{f(x+h) - f(x)}{h}.$$

Se considera ahora una función $f : \mathcal{F}_\mathcal{H}(\mathbb{U}) \to \mathcal{F}_\mathcal{H}(\mathbb{U})$ constante $f(x) = A \in \mathcal{F}_\mathcal{H}(\mathbb{U})$, donde A es un número difuso dado, en este caso la diferencia se podría definir como:

$$[f(x+h) - f(x)] = [A - A].$$

Sean $[A]_\alpha$ los α-cortes de A entonces, para todo $h > 0$:

$$[f(x+h) - f(x)]_\alpha = [A-A]_\alpha = [a_\alpha^- - a_\alpha^+, a_\alpha^+ - a_\alpha^-],$$

y por tanto, cuando h tiende a 0, el soporte de

$$\frac{[f(x+h) - f(x)]_\alpha}{h},$$

tiende a infinito, luego no es compacto, por tanto

$$\lim_{h\to 0} \frac{[f(x+h) - f(x)]_\alpha}{h} \notin \mathcal{F}_\mathcal{H}(\mathbb{U}).$$

Este problema nos motivará a tratar un nuevo concepto de diferencia, la diferencia de Hukuhara.

2.4.3. Hukuhara y diferencia generalizada

Debido al problema especificado en la sección anterior, se necesita definir una operación resta que cumpla que $A - A = \{0\}$, Hukuhara dio una definición de resta que soluciona el problema expuesto en la sección anterior.

Definición 8. *Dados dos números difusos $A, B \in \mathcal{F}_{\mathcal{C}}(\mathbb{R})$ la diferencia de Hukuhara (H-Diferencia) se define como $A \ominus_H B = C$ donde C es el número difuso que cumple $A = B + C$, si existe.*

Observación 1. *Sean $A, B, C \in \mathcal{F}_{\mathcal{C}}\mathbb{R}$ se consideran sus α-cortes, $[A]_\alpha = [a_\alpha^-, a_\alpha^+]$, $[B]_\alpha = [b_\alpha^-, b_\alpha^+]$ y $[C]_\alpha = [c_\alpha^-, c_\alpha^+]$.*
De donde,

$$[a_\alpha^-, a_\alpha^+] = [b_\alpha^- + c_\alpha^-, b_\alpha^+ + c_\alpha^+].$$

Por tanto,

$$[A \ominus_H B]_\alpha = [a_\alpha^- - b_\alpha^-, a_\alpha^+ - b_\alpha^+].$$

Con esta observación es fácil ver que la H-Diferencia cumple que $A - A = \{0\}$
Se pueden definir otras diferencias que mejoran el concepto anterior:

Definición 9 (Diferencia de Hukuhara generalizada)**.** *Dados dos números difusos $A, B \in \mathcal{F}_{\mathcal{C}}(\mathbb{R}$ se define la diferencia generalizada de Hukuhara (gH-diferencia) $A \ominus_{gH} B = C$ donde C es un número difuso que existe y cumple una de las siguientes condiciones:*

1. $A = B + C$.
2. $B = A - C$.

Definición 10 (Diferencia generalizada)**.** *Dados dos números difusos $A, B \in \mathcal{F}_{\mathcal{C}}(\mathbb{R})$ se define la diferencia generalizada (g-diferencia) $A \ominus_g B = C$ donde C es un número difuso caracterizado por los siguientes $\alpha-$cortes que existe y tiene los siguientes α-cortes:*

$$[A \ominus_g B]_\alpha = cl \bigcup_{\beta \geq \alpha} ([A]_\beta \ominus_{gH} [B]_\alpha), \text{ para todo } \alpha \in [0, 1].$$

2.5. Interactividad

El principio de extensión de Zadeh se puede aplicar a funciones de distinto número de argumentos. Los ejemplos más simples podrían ser la suma, la resta, la multiplicación y la división de números difusos. Esta situación es más compleja, debido a que hay que tener en cuenta las estructuras entre los distintos argumentos. Esta dependencia mutua entre los distintos conjuntos difusos, viene dada por una función de pertenencia común denominada función de pertenencia conjunta. En términos de conjuntos difusos, esta dependencia se llama interactividad. Las definiciones de esta sección se toman de [4]

Definición 11 (Interactividad, función de pertenencia conjunta)**.** *Sea $\hat{a} \in \mathcal{F}(V)$ y sea $\hat{b} \in \mathcal{F}(W)$. Entonces la interactividad de $\hat{a}$ y $\hat{b}$ se define por la función de pertenencia conjunta: $\mu_{\hat{a},\hat{b}} : V \times W \to [0, 1]$*

Para calcular las funciones de pertenencia marginales, simplemente hay que aplicar el principio de extensión de Zadeh:

Definición 12 (Función marginal de una función de pertencia conjunta)**.** *Se define la función de pertenencia marginal respecto a como:*

$$\mu_a(a) = \sup_{b \in W} \mu_{\hat{a},\hat{b}}(a, b).$$

Se suele suponer no interactividad al trabajar con varios conjuntos difusos, esto es;

Definición 13 (No interactivos o independientes)**.** *Dos conjuntos difusos $\hat{a} \in \mathcal{F}(V)$ y $\hat{b} \in \mathcal{F}(W)$ se dicen que son no interactivos, o independientes si $\mu_{\hat{a},\hat{b}}(a, b) = \min(\mu_{\hat{a}}(a), \mu_{\hat{b}}(b))$*

2.6. Métrica en conjuntos difusos [1]

En esta sección se va a generalizar la definición de espacio métrico a conjuntos difusos, y vamos a dar algunos resultados importantes. Todas las demostraciones de esta sección pueden encontrarse en [3]. Se va a ver en primer lugar el concepto de métrica:

Definición 14 (Pseudométrica)**.** *Sean A y B dos subconjuntos compactos de un espacio métrico $\mathbb{U}$. Entonces se define la pseudométrica como:*

$$\rho(A, B) = \sup_{a \in A} d(a, B),$$

donde:

$$d(a, B) = \inf_{b \in B} ||a - b||,$$

es la separación de Haussdorf.

Definición 15 (Métrica de Pompeiu-Hausdorff)**.** *Sean A y B dos conjuntos difusos, un espacio métrico $\mathbb{U}$. La métrica de Pompeiu-Hausdorff, denotada por d_∞, se define como:*

$$d_\infty(A, B) = \sup_{\alpha \in [0,1]} \max\{\rho([A]_\alpha, [B]_\alpha), \rho([B]_\alpha, [A]_\alpha)\}.$$

Si A y B fueran números difusos, tendríamos:

$$d_\infty(A, B) = \sup_{\alpha \in [0,1]} \max\{|a_\alpha^- - b_\alpha^-|, |a_\alpha^+ - b_\alpha^+|\}.$$

Teorema 5 ([5])**.** *El espacio de los números difusos, con la métrica d_∞ es un espacio de Banach.*

2.7. Funciones difusas

Definición 16 (Función con valores en conjuntos difusos)**.** *Se dice que una función F es una función con valores en conjuntos difusos si:*

1. $dom\ F \subset \mathbb{R}$.
2. $Im\ F \subset \mathcal{F}_{\mathcal{H}}(\mathbb{R}^n)$.

Ejemplo 2. *La función $f(x) = Ax$ donde $A = [a, b]$ con $a, b \in \mathbb{R}$, función con valores en conjuntos difusos. Sus imágenes son intervalos.*

Ejemplo 3. *La función $f(x) = Ax$ donde $A = (a; b; c)$ con $a < b < c$, es una función con valores en conjuntos difusos. Además, su imágenes son conjuntos números triangulares.*

2.7.1. Continuidad de funciones difusas

Se introduce en primer lugar dos conceptos de continuidad sobre funciones reales que toman valores en subconjuntos reales. Y luego, usaremos esta misma idea para definir la continuidad sobre conjuntos difusos

Definición 17 (Función con valores en conjuntos difusos continua)**.** *Sea $F : \Omega \to \mathcal{P}(\mathbb{R}^n)$, $\Omega \subset \mathbb{R}^m$ se dice que F es semicontinua superiormente en $t_0 \in \Omega$, si para todo $\varepsilon > 0$ existe un $\delta > 0$ tal que:*

$$\rho(F(t), F(t_0)) < \varepsilon.$$

Si $||t - t_0|| < \delta$ para $t \in \Omega$.
Por otro lado, se dice que F es semicontinua inferiormente en $t_0 \in \Omega$, si para todo $\varepsilon > 0$ existe un $\delta > 0$ tal que:

$$\rho(F(t_0), F(t)) < \varepsilon.$$

Si $||t - t_0|| < \delta$ para $t \in \Omega$.
Si una función es semicontinua superiormente e inferiormente, se dice que es continua.

Definición 18 (Función difusa continua)**.** *Sea $F : \Omega \to \mathcal{F}_{\mathcal{H}}(\mathbb{R}^n)$, $\Omega \subset \mathbb{R}^m$ se dice que F es semicontinua superiormente en $t_0 \in \Omega$, si para todo $\varepsilon > 0$ existe un $\delta > 0$ tal que:*

$$\rho([F(t)]^{\alpha}, [F(t_0)]^{\alpha}) < \varepsilon.$$

Si $||t - t_0|| < \delta$ para $t \in \Omega$, para todo $\alpha \in [0, 1]$.
Por otro lado, se dice que F es semicontinua superiormente en $t_0 \in \Omega$, si para todo $\varepsilon > 0$ existe un $\delta > 0$ tal que:

$$\rho([F(t_0)]^{\alpha}, [F(t)]^{\alpha}) < \varepsilon.$$

Si $||t - t_0|| < \delta$ para $t \in \Omega$, para todo $\alpha \in [0, 1]$.
Si una función es semicontinua superiormente e inferiormente, se dice que es continua.

3 ECUACIONES DIFERENCIALES DIFUSAS

"Es imposible ser matemático sin ser un poeta del alma"

– Sofia Kovalévskaya

En este capítulo se van a explorar los diferentes puntos de vista a los conceptos clásicos del cálculo mediante una perspectiva difusa. Se va a recordar la definición de integral de Riemann, Aumann & Henstock, y haremos una revisión de la definición de derivada de Hukuhara (Diferencia de Hukuhara). Además, se tratará de la versión difusa del teorema fundamental del cálculo que nos ayudará a tener una mejor visión de la teoría estudiada.

3.1. Cálculo difuso para funciones definidas en conjuntos difusos

Esta sección está basada en el contenido de [1].

3.1.1. Distintas definiciones de derivada

En primer lugar, vamos a definir los distintos conceptos de derivada difusa.

La derivada de Hukuhara

La derivada de Hukuhara está basada en el concepto de diferenciabilidad de Hukuhara para funciones evaluadas en intervalos ([6])

Definición 19 (Diferenciable según Hukuhara)**.** *Sea F una función definida en un conjunto difuso. Se supone que que los límites:*

$$\lim_{h\to 0^+} \frac{F(x_0+h)\ominus_H F(x_0)}{h} \quad \Big\| \quad \lim_{h\to 0^+} \frac{F(x_0)\ominus_H F(x_0-h)}{h} \quad .$$

Existen, y son iguales a cierto elemento $F'_H(x_0) \in \mathcal{F}_{\mathcal{H}}(\mathbb{R}^n)$, entonces F es diferenciable según Hukuhara (H-Diferenciable) en x_0 y se dice que $F'_H(x_0)$ es su derivada en x_0

Ejemplo 4 (Función constante). *Sea $F(x) = A$ con $A = (-1; 0; 1)$, se va a calcular su H-Derivada en un punto arbitrario $x_0 \in \mathbb{R}$:*

$$F(x_0 + h) \ominus_H F(x_0) = A \ominus_H A = 0.$$

$$F(x_0) \ominus_H F(x_0 - h) = A \ominus_H A = 0.$$

Por tanto, $F'_H(x_0) = 0$

Ejemplo 5 (Función lineal). *Sea $F(x) = Ax$ con $A = (-1; 0; 1)$, se supone en primer lugar que $x \geq 0$:*

$$F(x + h) \ominus_H F(x) = A(x + h) \ominus_H A = (-x - h; 0; x + h) \ominus_H (-x; 0; x) = (-h; 0; h).$$

Análogamente,

$$F(x) \ominus_H F(x - h) = (-h; 0; h),$$

de donde,

$$\lim_{h\to 0^+} \frac{(-h;0;h)}{h} = A \;\Big\|\; \lim_{h\to 0^+} \frac{(-h;0;h)}{h} = A \ .$$

Por tanto, $F'_H(x) = A$, si $x > 0$.
Por otro lado, si $x < 0$, se puede ver que $F(x + h) \ominus_H F(x)$ no está definido, pues:

- *$(-x - h; 0; x + h)$ no sería un número triangular, pues $-x - h > x + h$*

Esto se puede ver más fácilmente en un resultado que se muestra a continuación.

Proposición 1 (Construcción de funciones H-Diferenciables [7]). *Sea G una función definida en conjuntos difusos tal que $G(x) = Bg(x)$ donde $g(x) > 0, g'(x) > 0$ y B es un número difuso entonces, $G(x)$ es H-Diferenciable, más aún,*

$$G'_H(x) = Bg'(x).$$

Teorema 6 (Continuidad de funciones H-Diferenciables [8]). *Sea $F : [a, b] \to \mathcal{F}_{\mathcal{H}}(\mathbb{R}^n)$ una función H-Diferenciable, entonces F es continua en el sentido visto en el Tema 1.*

Teorema 7 (Álgebra en derivadas [8]). *Sean $F, G : [a, b] \to \mathcal{F}_{\mathcal{H}}(\mathbb{R}^n)$ funciones diferenciables y sea $\lambda \in \mathbb{R}$. Entonces, $(F + G)'_H = F'_H + G'_H$ y $(\lambda F)'_H = \lambda F'_H$.*

Una función H-Diferenciable tiene α-corte diferenciables, sin embargo, que todos los α-corte sean diferenciables, no implica que la función sea H-diferenciable.

La derivada de Seikkala

Ahora se introduce el concepto de derivada de Seikkala, que se usará después para introducir un concepto de derivada más fuerte.

Definición 20 (Derivada de Seikkala)**.** *Sea* $F : [a, b] \to \mathcal{F}_H(\mathbb{R})$ *si:*

$$[(f_\alpha^-)'(x_0), (f_\alpha^+)'(x_0)].$$

existe para cada $\alpha \in [0, 1]$ *y define* α*-cortes de un número difuso* $F'_S(x_0)$ *entonces se dice que* F *es Seikkala diferenciable en* x_0 *y definimos* $F'_S(x_0)$ *como la derivada de* F *en* x_0*.*

A continuación, se introduce una condición necesaria que servirá de herramienta para relacionar el concepto anteriormente visto de derivada y la derivada de Seikkala.

Teorema 8 (Condición necesaria Seikkala [8])**.** *Si* $F : [a, b] \to \mathcal{F}_{\mathcal{H}}(\mathbb{R}^n)$ *es H-Diferenciable entonces* $f_\alpha^-(x)$ *y* f_α^+ *son diferenciables y:*

$$[F'(x_0)]_\alpha = [(f_\alpha^-)(x_0), (f_\alpha^+)(x_0)].$$

Entonces, F *es diferenciable según Seikkala y la derivada de Seikkala y de Hukuhara coinciden.*

La derivada fuertemente generalizada

El concepto de derivada fuertemente generalizada viene a generalizar más aún los conceptos de derivadas ya introducidos por Hukuhara y Seikkala.

Uno de los problemas que nos encontramos con las derivada de Seikkala y Hukuhara es cuando pasa $(f_\alpha^-)'(x_0) \leq (f_\alpha^+)'(x_0)$ (Se puede ver en el Ejemplo 5). Si F es Seikkara diferenciable (en particular, si es Hukuhara diferenciable), entonces $(f_\alpha^-)'(x) \leq (f_\alpha^+)'(x)$ y por tanto la diferencia $(f_\alpha^+)'(x) - (f_\alpha^-)'(x) \geq 0$. Es decir, para todo α-corte, la función diámetro$[F(x)]_\alpha = f_\alpha^+(x) - f_\alpha^-(x)$ es no decreciente. Pero esto es un serio inconveniente: significa que no pueden existir funciones Hukuhara diferenciables que presenten un grado de incertidumbre cada vez menor, o bien cuyo grado de incertidumbre oscile (crezca y decrezca). Para evitar estos problemas, se introdujo el concepto de derivada fuertemente generalizada:

Definición 21 (Derivada fuertemente generalizada)**.** *Sea* $F : (a, b) \to \mathcal{F}_{\mathcal{H}}(\mathbb{R}^n)$*. Si alguno de los siguientes pares de límites:*

1. $$\lim_{h\to 0^+} \frac{F(x_0+h)\ominus_H F(x_0)}{h} \;\Big\|\; \lim_{h\to 0^+} \frac{F(x_0)\ominus_H F(x_0-h)}{h}.$$
2. $$\lim_{h\to 0^+} \frac{F(x_0+h)\ominus_H F(x_0)}{-h} \;\Big\|\; \lim_{h\to 0^+} \frac{F(x_0)\ominus_H F(x_0-h)}{-h}.$$
3. $$\lim_{h\to 0^+} \frac{F(x_0+h)\ominus_H F(x_0)}{h} \;\Big\|\; \lim_{h\to 0^+} \frac{F(x_0)\ominus_H F(x_0-h)}{-h}.$$
4. $$\lim_{h\to 0^+} \frac{F(x_0+h)\ominus_H F(x_0)}{-h} \;\Big\|\; \lim_{h\to 0^+} \frac{F(x_0)\ominus_H F(x_0-h)}{h}.$$

Existen y son iguales a algún elemento $F'_G(x_0)$ de $\mathcal{F}_{\mathcal{H}}(\mathbb{R}^n)$, entonces F es diferenciable fuertemente generalizado (o GH) en x_0, y $F'_G(x_0)$ es el valor de la derivada. Se denota como GH-Diferenciable.

De la definición es trivial demostrar que si H-Diferenciable entonces es diferenciable fuertemente generalizado.

Observación 2. *De la definición anterior se pueden extraer las siguientes conclusiones, basadas en la naturaleza del diámetro de las funciones difusas:*

1. *Concepto de derivada para funciones con diámetro no decreciente.*
2. *Concepto de derivada para funciones con diámetro no creciente.*
3. *Concepto de derivada para funciones con diámetro con monotonía arbitraria.*
4. *Concepto de derivada para funciones con diámetro con monotonía arbitraria.*

Otras definiciones

Para cerrar esta sección, veremos dos conceptos más derivada:

Definición 22 (Diferencial de Hukuhara generalizada)**.** *Sea $F : (a, b) \longrightarrow \mathcal{F}_{\mathcal{C}}(\mathbb{R})$. Si el límite:*

$$\lim_{h\to 0} \frac{F(x_0 + h) \ominus_{gH} F(x_0)}{h},$$

existe y pertenece a $\mathcal{F}_{\mathcal{C}}(\mathbb{R})$, entonces F es diferenciable de forma generalizada según Hukuhara (gH diferenciable) en x_0, además a $F'_{gH}(x_0)$ se le llama el valor de la derivada de forma generalizada según Hukukahara

Definición 23 (Diferencial generalizada)**.** *Sea $F : (a, b) \longrightarrow \mathcal{F}_{\mathcal{C}}(\mathbb{R})$. Si el límite:*

$$\lim_{h\to 0} \frac{F(x_0 + h) \ominus_{g} F(x_0)}{h},$$

existe y pertenece a $\mathcal{F}_{\mathcal{C}}(\mathbb{R})$, entonces F es diferenciable de forma generalizada (g diferenciable) en x_0, además a $F'_{g}(x_0)$ se le llama el valor de la derivada de forma generalizada

Para las definiciones anteriores se puede demostrar la siguiente jerarquía de implicaciones:

H-Diferenciable $\Rightarrow$ GH-Diferenciable $\Rightarrow$ gH-Diferenciable $\Rightarrow$ g-Diferenciable

La demostración de lo anterior, como el resto de afirmaciones del capítulo pueden encontrarse en [1]

3.1.2. Integral

La primera propuesta de integral difusa se basa en el concepto integral de Aumann [9] para funciones multievaluadas. La definición original podemos verla en [10] y [11].
Por simplificar, vamos a denotar:

$$S(G) = \{g : I \to \mathbb{R}^n : \text{g integrable}, g(t) \in G(t), \forall t \in I\}.$$

Vamos a denotar, para cada $G : I \to \mathcal{P}(\mathbb{R}^n)$

Integral de Aumann

Definición 24 (Integral de Aumann)**.** *La integral de Aumann de una función sobre un conjunto difuso $F : [a, b] \to \mathcal{F}_C(\mathbb{R}^n)$ sobre $[a, b]$ se define como:*

$$\left[(A)\int_a^b F(x)dx\right]_\alpha = \left\{\int_a^b g(x)dx : g \in S([F(x)]_\alpha)\right\}.$$

Para todo $\alpha \in [0, 1]$. La función F se dirá que es integrable sobre $[a, b]$ si $(A)\int_a^b F(x)dx \in \mathcal{F}_{\mathcal{H}}(\mathbb{R}^n)$

Integral de Riemann

Se puede usar también el concepto clásico de integral para definir una integral de Riemann en el ámbito difuso.

Definición 25 (Integral de Riemann)**.** *La integral de Riemman de una función difusa $F : [a, b] \to \mathcal{F}_{\mathcal{C}}(\mathbb{R})$ sobre $[a, b]$ es el número difuso A tal que para todo $\varepsilon > 0$ existe un $\delta > 0$ tal que para cualquier partición $\mathcal{P} := a = x_0 < x_1 < ... < x_n = b$ con $x_i - x_{i-1} < \delta, i = 1, ..., n$ y $\xi_i \in [x_i - x_{i-1}]$*

$$d_\infty\left(\sum_{i=1}^{n-1} F(\xi_i)(x_i - x_{i-1}, A\right) < \varepsilon.$$

La función F se dice que es integrable según Riemann sobre $[a, b]$ y se denota:

$$(R)\int_a^b F(x)dx = A.$$

Algunos resultados importantes

En esta sección mostraremos una serie de teoremas de utilidad encontrados en [11] que reproducen en cierta manera algunas propiedades ya conocidas del cálculo infinitesimal clásico:

Teorema 9. *Si una función $F : [a, b] \to \mathcal{F}_{\mathcal{H}}(\mathbb{R})$ es continua según (Definición 17) entonces es integrable. Más aún,*

$$\left[\int F\right]_\alpha = \left[\int f_\alpha^-, \int f_\alpha^+\right],$$

para todo $\alpha \in [0, 1]$

Teorema 10 (Cambio de intervalos)**.** *Sea $F : [a, b] \to \mathcal{F}_{\mathcal{H}}(\mathbb{R})$ una función integrable, y supongamos $a \leq x_1 \leq x_2 \leq x_3 \leq b$, entonces:*

$$\int_{x_1}^{x_3} F = \int_{x_1}^{x_2} F + \int_{x_2}^{x_3} F.$$

Teorema 11. *Sean $F, G : [a, b] \to \mathcal{F}_{\mathcal{H}}(\mathbb{R})$ funciones integrables, entonces:*

1. $\int F + G = \int F + \int G$.
2. $\int \lambda F = \lambda \int F$ *para cualquier* $\lambda \in \mathbb{R}$.
3. $d_\infty(F, G)$ *es integrable.*
4. $d_\infty(\int F, \int G) \leq \int d_\infty(F, G)$.

3.1.3. Teorema fundamental del cálculo

En la teoría clásica del análisis, el teorema fundamental del cálculo nos ofrece una visión bastante fuerte que relaciona las derivadas y las integrales. En el cálculo difuso tenemos lo mismo, y los siguientes teoremas que vamos a introducir nos generalizarán estos teoremas clásicos para casos difusos.

Teorema 12 (Teorema fundamental del cálculo difuso [11])**.** *Sea $F : [a, b] \to \mathcal{F}_{\mathcal{H}}(\mathbb{R}^n)$ una función continua, entonces $G(x) = \int_a^x F(s)ds$ es H-Diferenciable y además,*

$$G'_H(x) = F(x).$$

Teorema 13 ([11])**.** *Sea $F : [a, b] \to \mathcal{F}_{\mathcal{H}}(\mathbb{R}^n)$ una función H-Diferenciable y sea F'_H su derivada. Supongamos que F'_H es integrable sobre $[a, b]$. Entonces,*

$$F(x) = F(a) + \int_a^x F'_H(s)ds,$$

para todo $x \in [a, b]$.

También se puede enunciar un teorema un teorema parecido al anterior, pero esta vez exigiendo que la función es diferenciable fuertemente generalizada (2):

Teorema 14 ([12])**.** *Sea $F : [a, b] \to \mathcal{F}_{\mathcal{H}}(\mathbb{R}^n)$ una función diferenciable fuertemente generalizada (2) y sea F'_H su derivada. Supongamos que esta es integrable sobre $[a, b]$. Entonces,*

$$F(x) = F(a) + \int_a^x F'_H(s)ds,$$

para todo $x \in [a, b]$.

3.2. Ecuaciones diferenciales difusas

Del mismo modo que hay varios acercamientos al concepto de derivada, no podía ser menos el concepto de ecuación diferencial difusa. En esta sección vamos a tratar los distintas formas de acercarse a los conceptos de ecuación diferencial difusa. Esta sección toma como referencia [13]

3.2.1. Introducción

En esta sección vamos a empezar planteando un problema de valor inicial difuso de la misma manera que se introducen las ecuaciones diferenciales clásicas. Dado un número difuso X_0, se trata de hallar una función $X : [0, T] \to F_H(U)$ tal que:

$$X'(t) = f(t, X(t)),\ X(0) = X_0, \tag{3.1}$$

donde $f : [O, T] \times \mathcal{F}_{\mathcal{H}}(\mathbb{U}) \to \mathcal{F}(\mathbb{R}^n)$ se obtiene mediante el principio de extensión de Zadeh aplicada a una función continua $g : [0, T] \times U \to \mathbb{R}^n$. Además, f es continua por ser g continua, aplicando Teorema 3 se puede concluir:

$$[f(t, X)]_\alpha = g(t, [X]_\alpha).$$

Se puede asociar a esta ecuación diferencial difusa una ecuación diferencial ordinaria:

$$x'(t) = g(t, x(t)),\ x(0) = c. \tag{3.2}$$

Así, decimos que $X(t)$ es una solución difusa de (3.1) en (el sentido del principio de extensión) si existe una solución $x(t, c)$ de (3.2) y, para cada t fijo, se tiene que

$$X(t) = \hat{x}(t, X_0),$$

donde $\hat{x}(t, X_0)$ es el resultado de aplicar el principio de extensión de Zadeh respecto al parámetro c a la función x(t,c).

3.2.2. Teorema de equivalencia entre EDO y EDD

A continuación, se ofrece un teorema que dará una relación entre ecuaciones diferenciales difusas y ordinarias.

Teorema 15 (Equivalencia entre EDO y EDD [14])**.** *Sea U un conjunto abierto sobre $\mathbb{R}^n$ y supongamos que $[X_0]_\alpha \subset U$ con $\alpha \in [0, 1]$. Sea g una función continua, y suponga que para cada $c \in U$ existe una única solución $x(\cdot, c)$ del problema (3.2) y también que $x(t, \cdot)$ es continúa para todo $t \in [0, T]$ fijado. Entonces, existe una única solución difusa $X(t) = \hat{x}(t, x_0)$ del problema (3.1)*

El resultado anterior es muy potente, permitirá utilizar métodos numéricos ya conocidos para resolver ecuaciones diferenciales difusas fácilmente.
Para ver la potencia del Teorema anterior, se introducirán una serie de ejemplos:

Ejemplo 6. *Se plantea la siguiente ecuación diferencial difusa con valores iniciales:*

$$\left\{ \begin{array}{rcl} X' & = & -X(t) \\ X(0) & = & C \end{array} \right. ,$$

donde C representa un número difuso arbitrario. A continuación se construye el problema ordinario asociado al problema difuso de la misma manera que se hace al inicio de esta sección;

$$\left\{ \begin{array}{rcl} x' & = & -x(t) \\ x(0) & = & c \end{array} \right. .$$

Es bien conocido que este problema de valores iniciales tiene la siguiente solución:

$$x(t, c) = ce^{-t}.$$

Por tanto, dado que $x(t, c)$ es continúa para cada $t, c \in \mathbb{R}$ se puede aplicar el Teorema de equivalencia y tenemos que la solución del problema difuso es:

$$X(t) = C \cdot e^{-t}.$$

Ejemplo 7. *Se plantea ahora un problema un tanto más interesante:*

$$\left\{ \begin{array}{rcl} X' & = & X^2(t) \\ X & = & C \end{array} \right. ,$$

donde esta vez C es un número difuso triangular:

$$C(y) = \left\{ \begin{array}{ll} 3 - y & si \quad 2 \leq y \leq 3 \\ y - 1 & si \quad 1 \leq y \leq 2 \\ 0 & \end{array} \right. .$$

El problema ordinario (o determinístico asociado) viene dado por:

$$x'(t) = x^2(t), \; x(0) = c$$

y la solución de este problema viene dado por:

$$x(t, c) = \frac{c}{1 - tc},$$

para cada $t \in [0, \frac{1}{3})$ fijado, la función (x, t) es continua respecto a c, entonces se puede aplicar el Teorema de equivalencia y podemos concluir que existe una única solución difusa dada por $X(t) = \hat{x}(t, X_0)$. Se observa que es posible calcular la solución difusa aplicando directamente el principio de extensión de Zadeh y el Teorema 3. Por tanto, se pueden calcular los α-cortes de la siguiente forma para cada $\alpha \in [0, 1]$:

$$\begin{array}{rcl} [X(t)]_\alpha & = & [\hat{x}(t, X_0)]_\alpha \\ & = & x(t, [X_0]_\alpha) \\ & = & x(t, [1 + \alpha, 3 - \alpha]) \\ & = & [x(t, 1 + \alpha), x(t, 3 - \alpha)] \\ & = & \left[\frac{1+\alpha}{1-t-t\alpha}, \frac{3-\alpha}{1-3t+t\alpha}\right] \end{array} .$$

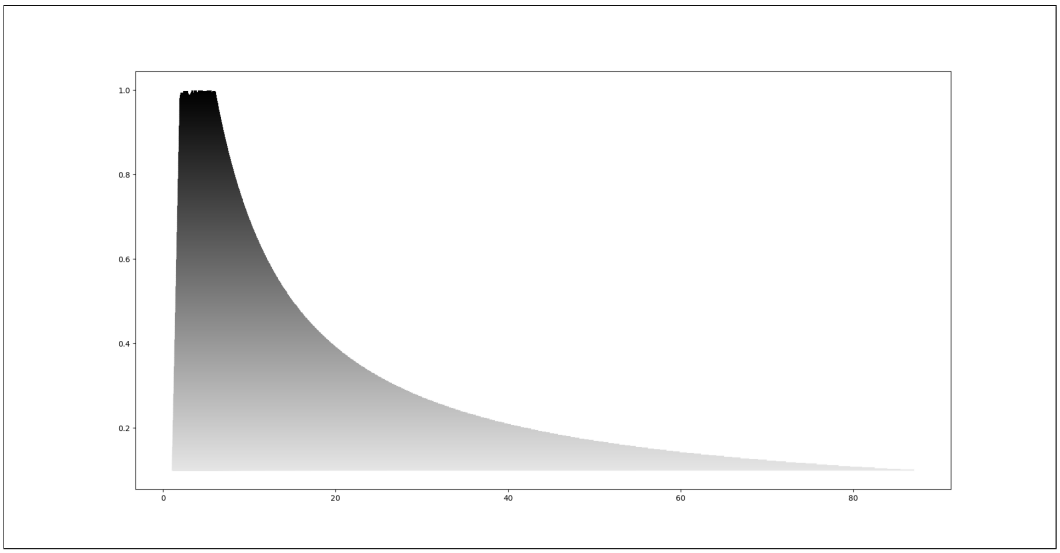


Figura 3.1: Gráfica de función de pertenencia

3.2.3. Inclusiones diferenciales

Esta sección está basada en la teoría que se puede encontrar en [1], que a su vez se fundamenta en los artículos [15], [16] y [17].
En estos artículos Hüllermeier y Diamond interpretan las Ecuaciones Diferenciales Difusas como una familia inclusiones diferenciales:

$$y'_\alpha(t) = g(t, y_\alpha(t)),\ y_\alpha(0) \in [X_0]_\alpha,\ 0 \leq \alpha \leq 1. \tag{3.3}$$

Bajo unas determinadas hipótesis, podemos afirmar:

$$\mathcal{A}_\alpha = \{y_\alpha | y_\alpha \text{ es una solución de la Familia de inclusiones diferenciales}\},$$

son los α-cortes de un determinado conjunto difuso y se le llaman soluciones del problema dado por la Familia de inclusiones diferenciales
A continuación se ofrece un teorema análogo al teorema que se vio en la sección introductoria Teorema de Equivalencia entre EDO y EDF

Teorema 16 ([14]). *Sea U un conjunto abierto en $\mathbb{R}^n$ y sea $X_0 \in \mathcal{F}(U)$. Dada la función g continua, para cada $c \in U$ existe una única solución $x(\cdot, c)$ del problema 3.2 y que $x(t, \cdot)$ es continúa en U para cada $t \in [0, T]$. Entonces, la solución difusa del problema (3.1) (en el sentido del principio de extensión de Zadeh) y las soluciones del del problema de Familia de inclusiones diferenciales coinciden, es decir;*

$$X(t) = \mathcal{A}(t),$$

para todo $t \in [0, T]$.

3.3. Ecuaciones en derivadas parciales

Hasta aquí toda nuestra discusión se ha enfocado en tratar de desarrollar una teoría para plantear y resolver ecuaciones diferenciales difusas.
En esta sección se va a desarrollar un método para resolver ecuaciones en derivadas parciales con condiciones de contorno difusas.

3.3.1. Método de líneas (MOL) [28]

Para desarrollar este método, se va a exponer un ejercicio.
Se considera el siguiente problema, ligado a una ecuación parabólica, con $x \in [0,1]$ y $t > 0$

$$\frac{\delta u(x,t)}{\delta t} = \frac{\delta^2 u(x,t)}{\delta x^2}$$

Donde debido a las limitaciones de nuestros instrumentos de medidas, no somos capaces de dar unas condiciones de contornos exactas, y consideramos un número difuso ϵ para representar este error en la medida. Y definimos las condiciones de contornos de la siguiente forma:

$$u(x,0) = \epsilon \sin \pi x$$
$$u(0,t) = u(1,t) = 0$$

Desarrollo del método de líneas

Consideramos la discretización de $[0,1]$ dada por $\mathcal{P}_N = \left\{ \frac{n}{N} : n \in \mathbb{Z}, 0 \leq n \leq N \right\}$ y $h = \frac{1}{N}$.
Definimos (x_n, t) con $x_n \in \mathcal{P}_N$ y $t > 0$ para algún $N > 0$ como la semirrecta vertical. Consideramos en esta recta una función de $t > 0$ como:

$$u_n(t) = u(x_n, t)$$

Observemos que por las condiciones de contorno tenemos:

$$u_0(t) = u(0,t) = 0$$

$$u_N(t) = u(1,t) = 0$$

Observemos que hemos conseguido una discretización de una variable, pero sigue siendo continua para la variable t.
Consideremos ahora para un $t \in [0, +\infty]$

$$f : x \longrightarrow u(x,t)$$

Esta función cumple:

$$\frac{\delta u(x,t)}{\delta t} = f''(x)$$

En particular,

$$\frac{\delta u(x_n, t)}{\delta t} = f''(x_n)$$

Aproximamos $\frac{\delta u(x_n,t)}{\delta t}$ mediante:

$$\frac{\delta u(x_n, t)}{\delta t} = \frac{u(x_{n-1}, t) - 2u(x_n, t) + u(x_{n+1}, t)}{h^2} + \mathcal{O}(h^2)$$

Agrupando las soluciones escalares en el vector de funciones:

$$u(t) = (u_1(t), u_2(t), ..., u_{N-1}(t))$$

Teniendo en cuenta que las funciones $u_0(t)$ y $u_N(t)$ son condiciones de contorno, idénticamente nulas, podemos escribir lo siguiente:

$$u' = \frac{1}{h^2} Au$$

Donde A es la matriz:

$$A = \begin{bmatrix} -2 & 1 & 0 & \cdots \\ 1 & -2 & 1 & \cdots \\ 0 & 1 & -2 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Por otra parte, la condición inicial para u:

$$u(x, 0) = \epsilon \sin \pi x$$

Para $u_n(0)$ podemos escribir:

$$u_n(0) = u(x_n, 0) = \epsilon \sin \pi\, nh$$

Con este procedimiento hemos conseguido escribir nuestra ecuación en derivadas parciales como una ecuación diferencial ordinaria.
La resolución numérica de este problema se puede llevar a cabo mediante métodos clásicos como Runge-Kutta.

4 COMPUTACIÓN CIENTÍFICA DE ALTO RENDIMIENTO

> *"La idea detrás de los computadores digitales puede explicarse diciendo que estas máquinas están destinadas a llevar a cabo cualquier operación que pueda ser realizada por un equipo humano"*
>
> – Alan Turing

En los anteriores capítulos se han dado definiciones y técnicas para generalizar los conceptos clásicos del cálculos en un contexto difuso, no obstante el objetivo final de este trabajo es encontrar y desarrollar métodos numéricos para resolver problemas en ecuaciones diferenciales difusas.
En este capítulo, se tratan los problemas desde el punto de vista informático y se abordan distintas técnicas numéricas que se van a aplicar en los capítulos posteriores para atacar estos problemas de la forma más eficiente posible.
Este capítulo está basado en las notas que podemos ver en [18].

4.1. Conceptos básicos

Dentro de la computación científica, podemos distinguir una serie de conceptos esenciales, que tienen que ver en cierta medida con las partes que forman un ordenador. Todo el mundo que tiene un ordenador, seguro que conoce ciertas partes de su ordenador, memoria RAM, el procesador, tarjeta gráfica, tarjeta de red... pero, ¿Qué impacto tiene cada uno de estos elementos en el desarrollo de la computación científica?

4.1.1. Memoria RAM

Los programas que se ejecutan en un ordenador se alojan en la memoria RAM, y una vez alojados en la memoria RAM, el procesador se encarga de procesar las instrucciones y ejecuta el código del programa procesando lo que se conoce como *stack*. En resumen, la memoria

RAM es donde se almacenan las instrucciones que va a ejecutar el procesador, y todos los datos que generamos en nuestro sistema operativo.
La memoria RAM nos impone un límite de la cantidad de datos que puede estar en ejecución en un momento dado, y si se quiere obtener el máximo rendimiento posible, se tiene que evitar escribir más memoria RAM de la disponible, si no, el sistema operativo empezará a usar el disco duro para alojar información. Esta tecnología se le conoce como *swap*, y es mucho más lenta que la memoria RAM.
Dentro de un ordenador, se puede dividir la memoria RAM en dos grupos; la memoria RAM disponible para el procesador, y la memoria RAM disponible para la tarjeta gráfica. Esto es bastante importante, pues cuando se trabaja con la tarjeta gráfica, no se puede acceder a punteros alojados en la memoria RAM del procesador, por tanto, antes de acceder a esta información se debe copiar a la memoria RAM de la tarjeta gráfica. La operación de copiar datos entre la memoria de la tarjeta gráfica al procesador es bastante lenta, y es conocido que en este punto existe un cuello de botella.
Tener mucha memoria RAM en nuestra máquina también podría reducir el consumo energético, pues se reduciría el acceso al disco duro.

4.1.2. Procesador (CPU)

El procesador es la parte del ordenador que se encarga de interpretar las instrucciones que hemos generado con nuestro programa, y es también fundamental a la hora de conseguir un rendimiento óptimo de un programa.
Algunos de los aspectos a tener en cuenta para obtener un mejor rendimiento serían los hilos del procesador, y los GHz. Los hilos del procesador permiten ejecutar tareas de manera simultanea. Por ejemplo, si el procesador tiene 10 hilos, y se quiere sumar un vector de 10 elementos, se puede hacer que las 10 sumas que hacen falta para sumar el vector se hagan simultáneamente. En el caso de que se quieran hacer operaciones que dependan unas de otras podemos considerar un grafo árbol donde vamos procesando los nodos independientes de forma paralela y esperamos en los nodos dependientes hasta que todas las operaciones hayan sido realizadas.

4.1.3. Tarjeta gráfica (GPU)

Habitualmente, se piensa sólo en el mundo *gaming* cuando se habla de tarjetas gráficas, y pensamos que su única utilidad es para jugar a videojuegos en alta resolución. Sin embargo, son más útiles de lo que puede parecer.

En este caso, las gráficas podemos tratarlas de forma parecida a las CPU, sin embargo, la gran ventaja que ofrecen las GPU es que están construidas para procesar muchos datos simultáneamente, debido a que tienen muchos más hilos disponibles que la CPU. Pero por otro lado, cada hilo es más lento.

4.1.4. Tarjeta de Red (Internet/Intranet/Cluster)

Usar internet o intranet para trabajar con computación de alto rendimiento es una buena práctica. Esto es que cuando conectamos varios ordenadores, llamados nodo, y los coordinamos para realizar una tarea. A este concepto se les llaman *cluster de ordenadores*.

Es importante saber que la información a través de la tarjeta de red viaja más lento que por las otras vías, y hay otros factores a tener en cuenta, como la distancia física entre los ordenadores. Sin embargo, si queremos trabajar con muchísimos hilos es la mejor opción que existe, esto es, conectar muchos ordenadores para realizar simultáneamente las operaciones que se le manden.

Tratemos de dar un símil con la vida real para esclarecer el funcionamiento. Pongamos que somos una empresa que vendemos productos por internet, donde hay miles de usuarios que hacen pedidos de forma simultaneas y queremos vender nuestros productos a toda España. Para ello necesitaríamos trabajar con servicios de mensajería. Cada servicio de mensajería podemos interpretarlo como un nodo de nuestro clúster, donde cada servicio de mensajería funciona de forma independiente pero tienen un objetivo común, que es cumplir la tarea que nosotros como empresa le hemos marcado.

4.2. Técnicas de alto rendimiento

Una vez introducidos los conceptos básicos acerca de los distintos elementos de la computación científica, vamos a hablar de las distintas técnicas que podemos abordar para obtener un rendimiento lo más óptimo posible.

4.2.1. Programación de bajo nivel

El lenguaje de programación que utilicemos va a decantar la balanza en temas de rendimiento. Si queremos exprimir al máximo nuestros ordenadores no nos quedará más remedio que decantarnos por lenguajes de más bajo nivel como C++ o C. Al no ser lenguajes interpretados, como pasa por ejemplo con Python, es decir, nuestro código se ejecuta directamente en nuestra máquina.
En las siguientes pruebas, vamos a revisar cómo se comportan Python y C, con el mismo código, en términos de tiempo de ejecución y de uso de RAM: (Ver código en el apéndice)

- C es más eficiente a la hora de administrar la memoria RAM, al probar 10000000000 iteraciones, Python no puede reservar más memoria RAM, sin embargo, C es capaz de reservar la memoria necesaria sin ningún tipo de problema.
- Por otro lado, los tiempos de ejecución son más sorprendentes aún. Con tan solo 10 iteraciones, C es 5 veces más rápido que Python, con 1000000 iteraciones, C es 77 veces más rápido que Python y, finalmente, con 1000000000 iteraciones, C es 100 veces más rápido que Python. Aquí se ve la notable diferencia entre uno y otro. Si queremos

trabajar en computación científica, trabajar con lenguajes compilados como C debe ser nuestra primera elección siempre que la necesidad de rendimiento sea crítica. (Ver figura 4.1)

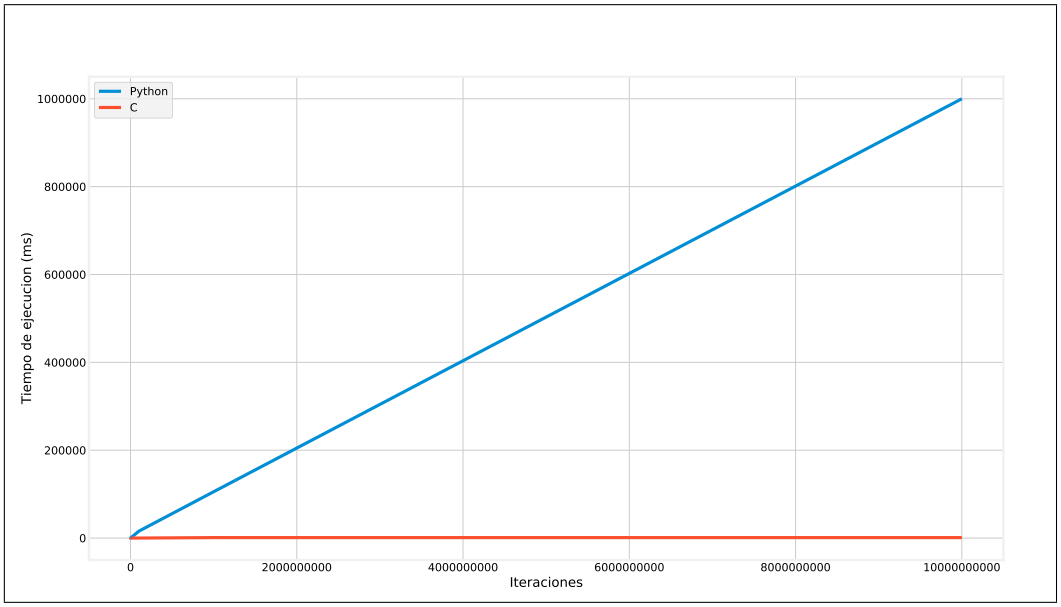


Figura 4.1: Gráfica comparativa C vs Python 8.1.1, 8.1.2

4.2.2. Precisión mixta

Otra técnica menos conocida, pero no por ello menos importante, es el uso de precisión mixta. Recordemos en primer lugar, que cuando trabajamos con números en un ordenador debemos de tener en cuenta la precisión con la que estamos trabajando. Existen algunas alternativas para trabajar con números en precisión *«infinita»*, sin embargo, no son de nuestro interés pues no rinden tan bien como queremos, así que nos centraremos en dos tipos de precisiones:

- Precisión simple: Los números se representan utilizando 4 bytes, por tanto podemos representar 256^4 combinaciones.
- Precisión doble: Los números se representan utilizando 8 bytes, por tanto, podemos representar 256^8 combinaciones.

Generalmente, los procesadores están optimizados para trabajar mejor con doble precisión. Las GPU suelen estar mejor optimizadas para trabajar con simple precisión, así que en estos casos, es útil tener en cuenta lo que llamamos precisión mixta si queremos tener un resultado en doble precisión. El procedimiento para trabajar con precisión mixta con un método iterativo es el siguiente:

- Planteamos en primer lugar nuestro problema de forma normal, y lo resolvemos realizando iteraciones en simple precisión.
- Una vez que tenemos el resultado en simple precisión, inicializamos nuestro problema con el valor obtenido, a continuación realizamos iteraciones usando doble precisión.

A continuación, mostramos un ejemplo ilustrativo aplicando el método de Newton usando precisión mixta

Ejemplo 8 (Método de Newton precisión mixta, comparativa y desarrollo)**.** *Supongamos que queremos encontrar las raices de la siguiente función:*

$$f(x) = (x-1)^8.$$

$$f'(x) = 8(x-1)^7.$$

Nuestro objetivo es intentar conseguir una precisión que sea el cero de la máquina, en el sentido de que la diferencia entre dos iteraciones sea cero. En primer lugar resolveremos el problema en simple precisión, aplicando el método de Newton habitual:

- *Tiempo de ejecución: 0m0,001s.*
- *Iteraciones en simple precisión: 99.*
- *Iteraciones en doble precisión: 0.*
- *Resultado: 0.99999815225601196289062500000.*

Lo resolvemos ahora para doble precisión:

- *Tiempo de ejecución: 0m0,001s.*
- *Iteraciones en simple precisión: 0.*
- *Iteraciones en doble precisión: 265.*
- *Resultado: 0.99999999999999955591079014993 7.*

Si ahora resolvemos el problema aplicando los principios de la precisión mixta:

- *Tiempo de ejecución: 0m0,001s.*
- *Iteraciones en simple precisión: 99.*
- *Iteraciones en doble precisión: 166.*
- *Resultado: 0.99999999999999955591079014993 7.*

Podemos observar, que al resolver nuestro problema con precisión mixta hemos reducido en 100 las operaciones que tenemos que realizar en doble precisión, obteniendo una mejora de rendimiento. El código se puede encontrar en 8.1.5, 8.1.6 y 8.1.7

4.2.3. Paralelización de algoritmos

Una de las técnicas más conocidas para acelerar las operaciones que realizamos con un ordenador, es paralelizar los procesos. Decimos que un algoritmo de n pasos se puede paralelizar si cada iteración es independiente y el valor devuelto de esa iteración no depende del resto de iteraciones.
Para conseguir la paralelización, podemos hacerlo mediante diferentes teécnicas:

- CPU: Utilizando los hilos disponibles en el procesador de nuestro ordenador. Todos los hilos son de alto rendimiento, pero la cantidad de hilos es bastante pequeña.
- GPU: Utilizando los hilos disponibles en la tarjeta gráfica de nuestro ordenador. Los hilos tienen un menor rendimiento que los de la CPU, sin embargo, la GPU contiene una gran cantidad de hilos.
- Red: Distribuimos el trabajo entre varios ordenadores a través de una conexión de red.
- Mixta: Cuando se mezclan distintas técnicas de paralelización, se dice que estamos trabajando en paralelización mixta.

A continuación, vamos a mostrar un ejemplo basado en el esquema de diferencias finitas de la ecuación del calor, a modo de entender mejor las mejoras que suponen cada uno:

Ejemplo 9 (Diferencias finitas: Secuencial VS. Paralelo VS. GPU [19])**.** *Planteamos el siguiente problema (ecuación del calor de 1-D): Dado un intervalo* $(a, b) \in \mathbb{R}$ *hallar*

$$u : (a, b) \times \mathbb{R} \to \mathbb{R},$$

tal que:

$$\frac{\delta u}{\delta t}(x, t) = \frac{\delta u}{\delta x}(x, t),$$

junto a condiciones de contorno e iniciales adecuadas.
Sean $u_{i,j} \in \mathbb{R}$ *aproximaciones de* $u(x_i, t_j)$ *para particiones* $\{x_i\}_{i=1}^{n}$ *y* $\{t_j\}_{j=1}^{m}$ *en espacio y tiempo. Entonces el esquema en diferencias finitas que vamos a tener en cuenta será:*

$$u_{i,j+1} = u_{i,j} + \mu(u_{i-1,j} - 2u_{i,j} + u_{i+1,j})$$

Podemos observar que los términos que aparecen a la derecha del esquema, dependen exclusivamente de términos de la etapa anterior, por tanto podemos paralelizar cada etapa. Podemos pasar entonces escribir el código

En esta prueba hemos escrito el mismo código en C, CUDA[1] *y hemos utilizado OpenMP*[2] *para generar una versión en paralelo de nuestro código inicial en C. Con esto conseguimos:*

[1] CUDA, es un lenguaje de programación basado en C++ que sirve para programar la GPU - https://www.nvidia.es/object/cuda-parallel-computing-es.html

[2] OpenMP es una API que nos ofrece el compilador para poder paralelizar algoritmos escritos en C/C++ fácilmente - https://www.openmp.org/

- *Tener un código sin paralelizar para poder comparar en C.*
- *Tener un código exactamente igual al anterior paralelizado en CPU con OpenMP.*
- *Tener un código parecido al anterior pero que funciona en paralelo en la GPU en CUDA.*

En la siguiente tabla, mostramos el tiempo de ejecución en segundos de cada uno de los programas utilizando las distintas técnicas: (Todas las pruebas son en simple precisión)

Mallado	C-No paralelo	CPU	GPU
5×5	0,001	0,001	0,441
25×25	0,001	0,001	0,441
50×50	0,001	0,001	0,441
70×70	0,001	0,001	0,441
100×100	0,001	0,001	0,484
1000×1000	0,005	0,005	0,472
10000×10000	1,178	0,624	0,997
20000×20000	8,961	7,754	2,594
22760×22760	22,99	9,227	3,221

A continuación, mostramos el consumo energético [3] *en* $\mu A/s$*:*

Mallado	C-No paralelo	CPU	GPU
5×5	0,1	0,23	57,33
25×25	0,1	0,23	57,33
50×50	0,1	0,23	57,33
70×70	0,1	0,23	57,33
100×100	0,1	0,23	62,92
1000×1000	0,5	1,15	61,36
10000×10000	117,8	143,52	129,61
20000×20000	896,1	1783,42	337,22
22760×22760	2299	2122,21	418,73

Cuadro 4.1: Medido con: power_app_stats

Podemos observar que trabajar con GPU nos ofrece un mejor rendimiento, tanto en términos de tiempo como económicos. El consumo energético es mucho menor en GPU que en CPU a lo largo del tiempo.

4.2.4. Optimizar compilación

Si se está trabajando con C y el compilador de GNU, GCC, uno de los compiladores de referencia para este lenguaje, se puede escribir la opción -Ofast a la hora de compilar nuestro programa. Se puede encontrar una discusión sobre esta opción entre desarrolladores de GCC y el propio Linus Torvarlds, creador de Linux. [20]
La opción anterior, incluye las optimizaciones que se hacen con -O3 y aparte, añade unas optimizaciones numéricas que son las que se van a explorar a continuación:

- *-fno-trapping-math/-fno-signaling-nans:* Esta opción hace que operaciones como dividir entre 0 no genere excepciones.
- *-fno-rounding-math/-fno-signed-zeros/-funsafe-math-optimizations:* Esta opción desactiva las propiedades aritméticas coma flotante, y las remplaza con las propiedades ordinarias en precisión infinita. Debido a esto y a errores de redondeo, con esta opción puede que $(x + y) + z \neq x + (y + z)$, y se diferenciarían en el error de redondeo.
- *-ffinite-math-only:* Desactiva las cantidades nan e inf, esto hace que internamente nuestro programa no tenga que buscar si aparecen nan o inf para controlar esas excepciones.
- *-fno-errno-math:* Desactiva la variable que contiene los errores al usar la librería matemática.

[3] power_app_stats es una aplicación desarrollada expresamente para esta memoria, que calcula la energía en uso comparando la descarga de la batería en sistemas Linux - https://github.com/JoseCarlosGarcia95/power_app_stats

- *-fcx-limited-range:* Desactiva la reducción al hacer la división compleja.

En el último capítulo mostraremos una implementación en la que se utiliza esta opción de compilación, aunque aquí se muestra un pequeño resumen de lo que supone la optimización en cuestión de tiempos en un ejemplo concreto, que se verá en el último capítulo.

Test	Tiempo
Fastmath simple precisión	11.9 segundos
Simple precisión	85 segundos
Fastmath doble precisión	12.15 segundos
Doble precisión	75.80 segundos

Además, también se consigue mejor consumo energético como se puede observar en la siguiente tabla

Test	Consumo energético
Fastmath simple precisión	$118{,}644028 \mu A/s$
Fastmath doble precisión	$118{,}644058 \mu A/s$
Simple precisión	$152{,}542389 \mu A/s$
Doble precisión	$169{,}491638 \mu A/s$

5 MODELOS DE ECUACIONES DIFERENCIALES DIFUSOS

"La vida es crecimiento, y cuanto más viajamos más verdad podemos comprender. Comprender las cosas que nos rodean es la mejor preparación para comprender las cosas que hay más allá."

– Hipatia de Alejandría

En este capítulo se van a construir modelos matemáticos teniendo en cuenta la incertidumbre presente en la realidad.

5.1. Aplicaciones a las ciencias naturales

En esta sección vamos a ver modelos matemáticos donde aparecen la incertidumbre de manera natural al tener en cuenta constantes físicas o errores de medidas.
Se puede observar en [21] los errores que se cometen al usar una serie de constantes astronómicas, este hecho será interesante para plantear una serie de problemas.

5.1.1. Aplicación a la mecánica clásica

Por otro lado, se va a plantear es la órbita de un planeta de manera clásica teniendo en cuenta ahora las constantes y su incertidumbre

Orbita planetaria

Ejemplo 10. *Se trata de encontrar la posición respecto al sol de un planeta en un tiempo t, para resolver este problema se usará Ley de la Gravitación Universal que dice:*

$$F_G = \frac{GMm}{r^2},$$

donde $G = 6{,}67424 \times 10^{-11} \pm 6{,}7 \times 10^{-15} m^3 kg^{-1} s^{-2}$ *y* M *es la masa del sol, que en este caso,* $M = 1{,}9884 \times 10^{30} \pm 2 \times 10^{26} kg$*, debido a la naturaleza de estas constantes podemos expresarlas como números triangulares, y de esta forma se convierte el problema clásico en un problema difuso.*
Por otro lado, se puede expresar r *como la distancia al sol, y dado que el sol se ha tomado como punto de referencia,* $r = \sqrt{x^2 + y^2}$ *donde* x, y *son las posiciones respecto a los eje* x, y *respectivamente.*
En otro orden de cosas, si se aplica la segunda ley de Newton, se sabe que $\vec{F} = m\vec{a}$*, por tanto se puede escribir la aceleración dada por la gravedad como*

$$\vec{a} = -\frac{GM}{r^2}(\cos\sigma, \sin\sigma)$$

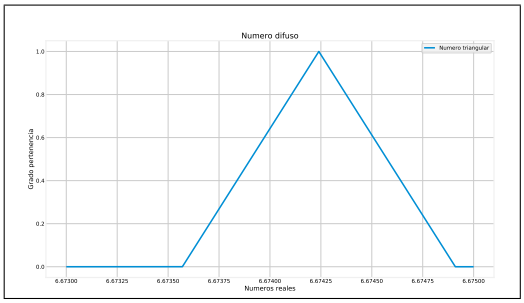
Y teniendo en cuenta la definición de seno, y coseno se puede escribir finalmente:

$$a_x = -\frac{GMx}{r^3} \;\Big\|\; a_y = -\frac{GMy}{r^3}$$

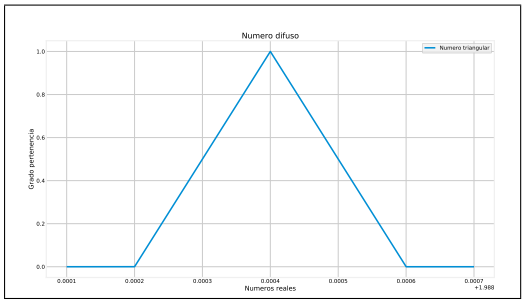
Y dado que por definición la aceleración es la segunda derivada de la ecuación de posición respecto al tiempo, se tiene:

$$\vec{a} = \frac{d\vec{s}}{dt}$$

Se definen ahora los números triangulares $G = (6{,}67357 \times 10^{-11}; 6{,}67424 \times 10^{-11}; 6{,}67491 \times 10^{-11})$ *y* $M = (1{,}9882 \times 10^{30}; 1{,}9884 \times 10^{30}; 1{,}9886 \times 10^{30})$*, se puede definir* $I = G \cdot M$*.*



(a) Gráfica del número difuso G



(b) Gráfica del número difuso M

Se puede observar también que tanto a_x, a_y *son funciones de clase* C^∞ *en el abierto* $\mathbb{R}^2 - (0, 0)$*, por tanto existe unicidad de soluciones (Teorema de Picard) y además, se puede aplicar el Teorema de equivalencia entre EDO y EDD, por tanto, se puede resolver este problema usando el método de Runge Kutta.*

5.1.2. Aplicación a la mecánica cuántica

La ecuación de Schröndinger

En mecánica cuántica, el principio de incertidumbre de Heisenberg son una serie de inecuaciones que nos dicen que no podemos medir la cantidad de partículas que hay en una cierta

región con precisión. Esta incertidumbre puede ser representada como condiciones de frontera difusa.
Este ejemplo se puede encontrar en [22]

5.1.3. Aplicación en modelos químicos

Esta sección está basada en el ejemplo que podemos encontrar en [23]

Desintegración radiactiva

Ejemplo 11. *Consideremos la ecuación de desintegración radiactiva dada por:*

$$x'(t) = -\lambda \cdot x(t),\ x(t_0) = x_0,$$

donde $x(t)$ es el radioisótopo presente en un material radiactivo, λ es la constante de desintegración y x_0 es el número inicial de radioisótopo. En este modelo, la incertidumbre aparece en el número inicial de radioisótopos teniendo en cuenta que el fenómeno de desintegración nuclear se considera como un proceso estocástico. Por tanto, si consideramos x_0 un número difuso, tendremos planteada nuestra ecuación diferencial difusa.

5.1.4. Aplicación en la medicina

También se ha aplicado la teoría difusa para modelar el crecimiento tumoral. Estos modelos podemos encontrarlos en la referencia dada en [24].

6

RESOLUCIÓN NUMÉRICA DE ECUACIONES DIFERENCIALES DIFUSAS

John Glenn: "Pasar de una trayectoria elíptica a otra parabólica"
Katherine Johnson: "No se trata de una solución teórica, sino numérica" [25]
– Conversación entre Katherine Johnson & John Glenn, *Figuras ocultas*

En esta sección se van a ver diferentes técnicas para resolver ecuaciones diferenciales difusas, abarcando desde las técnicas clásicas de la resolución númerica de ecuaciones diferenciales hasta las técnicas más sofisticadas a nivel computacional.
En esta sección se va a hacer un análisis exhaustivo de las ventajas en las ventajas que nos ofrecen los distintos métodos con vistas a obtener el rendimiento más óptimo en cuánto a velocidad y en eficiencia enerǵetica.
Para entender esta sección sería conveniente revisar algunas de las técnicas mostradas en la sección anterior, y repasar las tecnicas básicas de resolución numérica de ecuaciones diferenciales difusas.

6.1. El método de Euler

En esta sección se va a desarrollar el método de Euler para problemas difusos, para ello se empieza recordando como funciona el método de Euler para ecuaciones diferenciales ordinales. Cabe recordar que este método numérico tiene orden 1.

6.1.1. Método de Euler: Versión clásica

Se supone que se tiene un problema de valores iniciales bien definido, y lo suficiente regular para asegurar que tiene una única solución:

$$\begin{array}{rcl} y'(x) & = & f(x,y) \\ y(x_0) & = & y_0 \end{array}.$$

Se considera ahora una discretación del intervalo $[a, b]$ definida como:

$$x_i = x_0 + ih,$$

donde $h > 0$, $x_0 = a$.
El método de Euler define la siguiente solución aproximada: dada la condición inicial y_0

$$y_{i+1} = y_i + hf(x_i, y_i).$$

6.1.2. Método de Euler: Versión difusa

Se considera el siguiente problema difuso, al que se puede dar sentido para una función f(x,y) determinista utilizando el principio de extensión de Zadeh:

$$\begin{array}{rcl} y'(x) & = & f(x,y) \\ y(x_0) & = & A \end{array}',$$

donde A es un número difuso. Se supone que el problema anterior cumple el Teorema de equivalencia entre EDO y EDD, y se considera el problema determinista asociado:

$$\begin{array}{rcl} y' & = & f(x,y) \\ y(x_0) & = & a \end{array}$$

donde $a \in A$. Se toma entonces A como una discretización de $n+1$ elementos de A, donde cada elemento se denota por a_i, por tanto se tienen ahora $n+1$ EDO diferentes, donde cada una de estas se pueden resolver aplicando el método de Euler.

6.1.3. Método de Euler: Ejemplo

Ejemplo 12. *Sea un problema de valores iniciales difuso:*

$$y' = 2x - 3y + 1$$

$$y(0) = (-1; 0; 1).$$

Claramente cumple las hipótesis del Teorema de equivalencia entre EDO y EDD, por tanto el problema determinista asociado es:

$$y' = 2x - 3y + 1$$

$$y(0) = a,$$

con $a \in [-1, 1]$.
En primer lugar, hay que discretizar el intervalo $[-1, 1]$*, para ello, fijamos m>0 y para cada* $i = 0...m-1$ *se considera una partición dada por:*

$$a_i = -1 + \frac{2i}{m-1}.$$

Teniendo en cuenta estas particiones, se puede aproximar la solución del problema difuso resolviendo las m ecuaciones diferenciales que se deducen al tomar:

$$y' = 2x - 3y + 1$$

$$y(0) = a_i.$$

Se construye ahora el método de Euler para la ecuación asociada a a_i con tamaño del paso h, sea $x_0 = 0$ y tomemos $y_0 = a_i$, siguiendo entonces con le definición del método nos queda:

$$y_{j+1} = y_j + h(2x_j - 3y_j + 1)$$

$$x_{j+1} = x_0 + hj.$$

Para comparar el error de nuestro método, tengamos en cuenta que la solución exacta de la ecuación diferencial con dato inical a_i es:

$$y_i(x) = \frac{e^{-3x}(-1 + 9a_i + e^{3x}(1 + 6x))}{9}$$

A continuación, se ofrecerán varías implementaciones del método con información descriptiva acerca del rendimiento energético y en tiempo:

6.1.4. Experimentos numéricos

Se va a tratar de resolver el problema anterior aplicando el método de Euler, con una implementación en C y otra en Python, donde se varía la cantidad de particiones del número difuso triangular y con 100 pasos en el método de Euler.

Implementación en Python

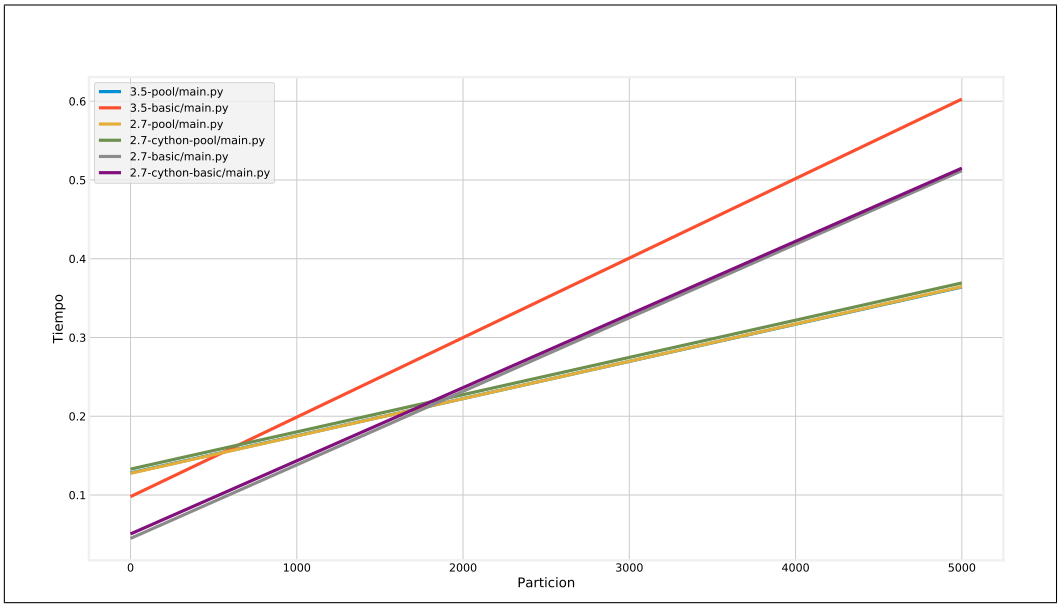


Figura 6.1: Distintos resultados en Python

Se pueden observar, varios patrones:

- *La forma menos eficiente de resolver el problema es usando una versión más reciente de Python.*
- *Trabajar con Cython no nos asegura un rendimiento mejor.*
- *El crecimiento del tiempo de ejecución en paralelo crece mucho menor que en secuencial.*
- *Si se quiere resolver un problema pequeño* $(n < 2000)$ *es conveniente hacerlo de manera secuencial.*

A continuación, se muestra una tabla con los resultados obtenidos por cada test.

Test	Tiempo
2.7 Paralelo	61 minutos
3.5 Paralelo	61 minutos
2.7 Cython	85 minutos
2.7 Secuencial	85.22 minutos
3.5 Secuencial	100 minutos

Implementación en C: Secuencial

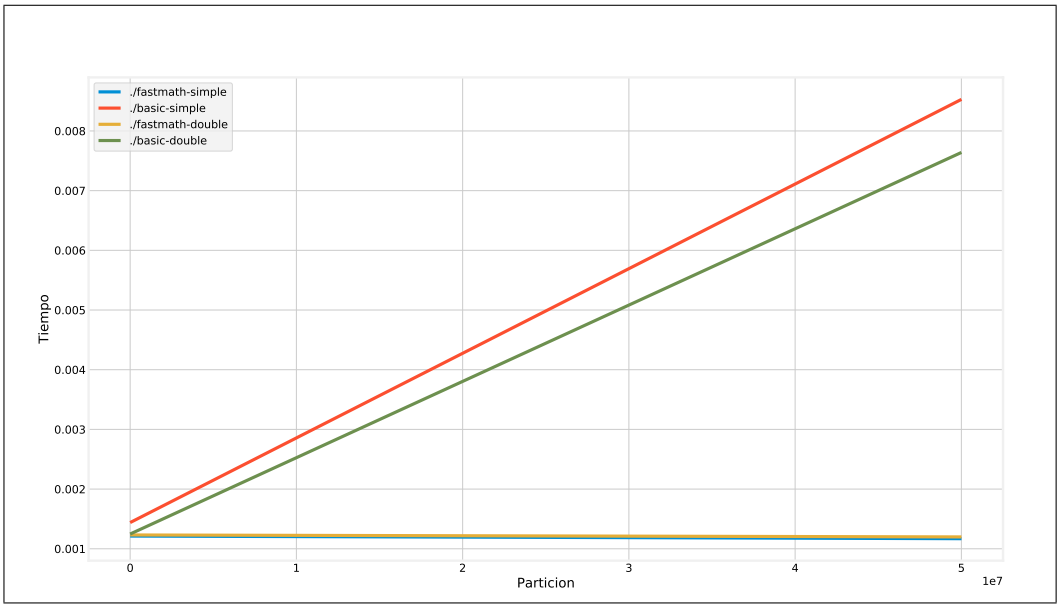


Figura 6.2: Distintos resultados en C

En esta parte se va a probar una implementación en C, usando diferentes técnicas numéricas.

- *Se han obtenido resultados esperados en las compilaciones básicas de los programas.*
- *Cuando se trabaja en doble precisión, el rendimiento se reduce, pero en C se siguen consiguiendo resultados bastantes rápidos independientemente de la precisión.*
- *El consumo de RAM es bastante reducido.*
- *El procesador al ser un procesador de 64 bits, trabaja mejor en doble precisión que en simple precisión.*

A continuación, se muestra una tabla con los resultados obtenidos por cada test.

Test	Tiempo
Fastmath simple precisión	11.9 segundos
Fastmath doble precisión	12.15 segundos
Doble precisión	75.80 segundos
Simple precisión	85 segundos

Ahora vamos a ver que sucede si también se aumenta el número de particiones para representar el número difuso, y para obtener resultados más interesantes, se van a empezar las iteraciones en 10000.

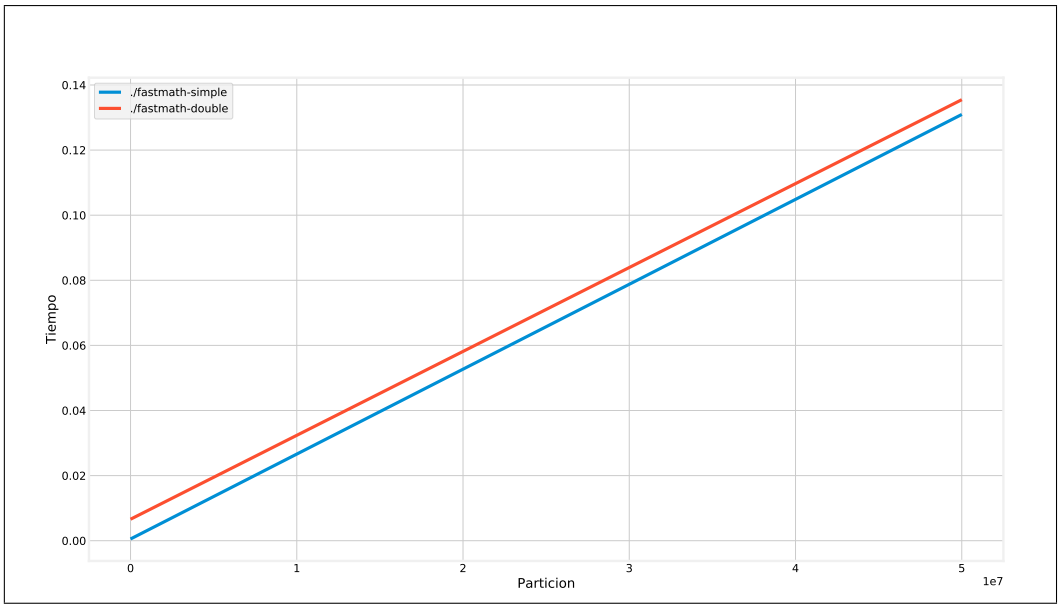


Figura 6.3: Distintos resultados en C

Aquí se pueden sacar una conclusiones diferentes,

1. *Es más óptimo trabajar en simple precisión con fastmath.*
2. *Pese a todo, aunque se esté haciendo* 10^{16} *iteraciones, el algoritmo tarda como mucho* $0{,}25$ *segundos, aún no es necesario plantearse trabajar en paralelo.*

Finalmente, se introduce una tabla con los tiempos de cada prueba

Test	Tiempo
Fastmath simple precisión	1278.03 segundos
Fastmath doble precisión	1341.10 segundos

Por otro lado, en la siguiente tabla se muestran los consumo energético de los distintos ejemplos para 10^5 *iteraciones.*

Test	Consumo energético
Fastmath simple precisión	$118{,}644028\mu A/s$
Fastmath doble precisión	$118{,}644058\mu A/s$
Simple precisión	$152{,}542389\mu A/s$
Doble precisión	$169{,}491638\mu A/s$

Conclusiones

En contra de lo que se pueda pensar comúnmente, trabajar con precisión doble es más eficiente en procesadores modernos de 64 bits que trabajar en precisión simple, así que no hay que tener

miedo a trabajar con doble precisión.
Los resultados obtenidos en C con la flag -OFast son sorprendentes, tan sorprendentes que no se ha planteado la necesidad de implementar el método en paralelo por la eficiencia de aplicar esa flag.
Y no sólo permite obtener una solución numérica más rápidamente, sino que permite consumir menos recursos energéticos, lo que se traduce en un ahorro económico a la hora de trabajar a una escala mayor, y un menor impacto medioambiental, por tanto utilizar la flag -Ofast al compilar es la mejor opción en todos los aspectos

6.2. Método de Runge-Kutta de cuarto orden

Por otro lado, vamos a desarrollar el método de Runge Kutta. Como es sabido el orden de convergencia de este método es mayor que el de Euler, por lo que se esperan mejores resultados a priori.

6.2.1. Método de Runge-Kutta de cuarto orden: Versión clásica

Se supone que se tiene un problema de valores iniciales bien definido, y lo suficiente regular para asegurar que tiene soluciones:

$$\begin{array}{rcl} y'(x) & = & f(x,y) \\ y(x_0) & = & y_0 \end{array}.$$

Como en el método de Euler, lo que se busca primer lugar es una discretización del intervalo $[x_0, x_f]$, se supone que se quiere discretizar el intervalo en n valores, sea entonces:

$$h = \frac{x_f - x_0}{n}.$$

Por tanto, podemos definir

$$x_i = x_0 + ih.$$

Por otro lado, el paso iterativo sobre y_{i+1} viene dado por:

$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4),$$

donde se definen:

$$\begin{array}{rcl} k_1 & = & f(x_i, y_i) \\ k_2 & = & f\left(x_i + \frac{h}{2}, y_i + \frac{k_1 h}{2}\right) \\ k_3 & = & f\left(x_i + \frac{h}{2}, y_i + \frac{k_2 h}{2}\right) \\ k_4 & = & f(x_i + h, y_i + k_3 h) \end{array}$$

6.2.2. Método de Runge-Kutta de cuarto orden: Versión difusa

Se considera el siguiente problema difuso, en el sentido del principio de extensión de Zadeh

$$\begin{array}{rcl} y' & = & f(x,y) \\ y(x_0) & = & A \end{array},$$

donde A es un número difuso. Se toma entonces que el problema anterior cumple el Teorema de equivalencia entre EDO y EDD, y se considera el problema determinista asociado:

$$\begin{array}{rcl} y' & = & f(x,y) \\ y(x_0) & = & a \end{array},$$

donde $a \in A$, se toma entonces A como una discretización de $n+1$ elementos de A, donde cada elemento se denota por a_i, por tanto se tienen ahora $n+1$ EDO diferentes, donde cada una de estas se pueden resolver aplicando el método de Método de Runge-Kutta de cuarto orden.

6.2.3. Método de Runge-Kutta de cuarto orden: Ejemplo

Para poder comparar correctamente los distintos métodos vamos a implementar el mismo problema usando el método de Runge-Kutta.

6.2.4. Experimentos numéricos

Se va a tratar de resolver el problema anterior aplicando el método de Runge-Kutta, con una implementación en C, donde se varía la cantidad de particiones del número difuso triangular, y con 100 pasos en el método de Euler.

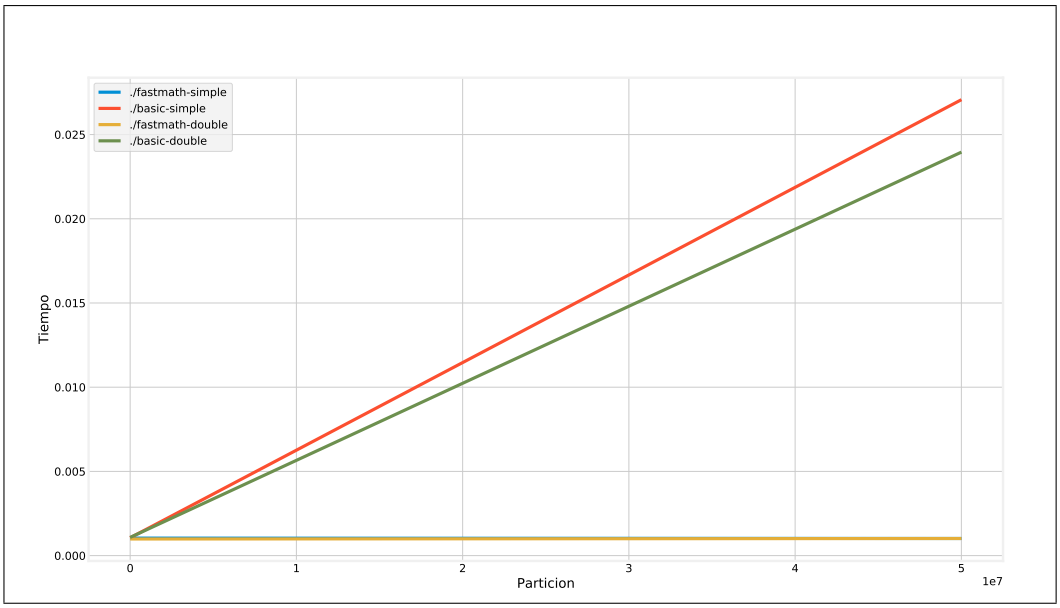


Figura 6.4: Distintos resultados en C

Test	Tiempo
Fastmath simple precisión	10.1921305656 segundos
Fastmath doble precisión	10.1441464424 segundos
Basic doble precisión	238.943392515 segundos
Basic simple precisión	270.01487565 segundos

6.3. Comparativas

Si se observa la tabla de tiempos del algoritmo de Euler respecto del algoritmo de Runge-Kutta se pueden encontrar una serie de informaciones interesante:

- Los modos básicos son más rápidos en el método de Euler, también tiene más sentido, pues en el método de Runge-Kutta necesita de más operaciones.
- Aunque Runge-Kutta precise de más operaciones, Runge Kutta tiene un mayor orden de convergencia.
- Al introducir la flag -Ofast el compilador optimiza Runge-Kutta lo suficiente como para poder afirmar que en este caso, Runge-Kutta es más rápido que el método de Euler, y además tiene un mayor orden de convergencia, por tanto, el mejor método en todos los aspectos es aparentemente el método de Runge-Kutta con la flag -ofast.

6.4. Método difuso paralelizado

Los dos métodos estudiados anteriormente se pueden resumir, de la siguiente forma:

- Se parte de una función determinista y se considera la ecuación difusa asociada a ella mediante el principio de extensión
- Se considera una discretización del número difuso.
- Se resuelve numéricamente para cada uno de los valores que se encuentra en la discretización.

Se observa, que el último paso es independiente para cada valor discretizado del número difuso, por tanto, se podría paralelizar el algoritmo anterior de manera que cada ecuación diferencial se resuelve al mismo tiempo de forma paralela, esto es útil si se quiere hacer una discretización muy fina del número difuso, podemos ver una implementación en 8.2.4

7 CONCLUSIONES

A lo largo de este trabajo se han ido generalizando los conceptos clásicos del análisis mediante definiciones difusas, y se ha encontrado que bajo ciertas condiciones se puede aplicar el Teorema de equivalencia entre EDO y EDD, que consigue hacer que resolver numéricamente una ecuación diferencial difusa sea equivalente a resolver una ecuación diferencial determinista para cada uno de los valores que toman los valores difusos.

Esto ofrece un amplio abanico de herramientas para resolver problemas de valores iniciales difusos, pues ahora la única complicación será tener la habilidad de poder resolver varios problemas.

Por tanto, las técnicas estudiadas aquí se aplican a todo problema en valores iniciales difuso que cumple las hipótesis del Teorema de equivalencia entre EDO y EDD, y por tanto se pueden resolver teniendo en cuenta todos los valores que toma el número difuso.

En resumidas cuentas, para resolver un problema de valores iniciales difusos se ha construido el siguiente procedimiento:

- Se parte de una función determinista y se considera la ecuación difusa asociada a ella mediante el principio de extensión.
- Se considera una discretización del número difuso.
- Se resuelve numéricamente para cada uno de los valores que se encuentra en la discretización.

También se ha observado que, mediante una serie de configuraciones a la hora de compilar el código, se puede obtener un menor consumo energético y un mejor rendimiento en cuestión de tiempo.

7.1. Planes de futuro

Una vez estudiadas las ecuaciones diferenciales difusas sería interesante dar un salto hacia las ecuaciones en derivadas parciales difusas, y ver si alguno de los resultados estudiados aquí se pueden generalizar.

Por otro lado, se podría tratar de implementar todo esto en un entorno de producción para ver las mejorías ecológicas y económicas que se nos ofrecen las técnicas computacionales aquí expuestas, y ver si en la práctica la optimización a nivel paranoico tiene el resultado esperado.

Finalmente, se podría trabajar con otro tipo de ecuaciones diferenciales y trabajar con ecuaciones en derivadas parciales como podemos ver en [26]. Sería muy interesante su aplicación a casos realistas, en modelos con aplicaciones en la ciencia o en la industria.

8 APÉNDICE: CÓDIGO FUENTE

8.1. Computación científica de alto rendimiento

8.1.1. Prueba simple en Python

```
#!/usr/bin/python
import sys

itera  = int(sys.argv[1])

vector = range(0, itera)

for i in range(0, itera):
    vector[i] = i + 1
```

8.1.2. Prueba simple en C

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char ** args) {
    int itera, i, *vector;

    itera = atoi(args[1]);

    vector = malloc(sizeof(int) * itera);

    for (i = 0; i < itera; i++) {
        vector[i] = i;
    }

    for (i = 0; i < itera; i++) {
        vector[i] = i + 1;
    }
}
```

8.1.3. Script de pruebas

```
#!/bin/bash

# Configuration

PY_EXEC="./prueba1.py"
C_EXEC="./prueba1"
RESULT="prueba1.log"

> $RESULT

for i in {1..10}; do
    ITERA=$((10 ** $i))

    echo "Python: Testing $ITERA"
    START_PYTHON=`date +%s%3N`
    $PY_EXEC $ITERA
    END_PYTHON=`date +%s%3N`

    echo "C: Testing $ITERA"
    START_C=`date +%s%3N`
    $C_EXEC $ITERA
    END_C=`date +%s%3N`

    RUNTIME_PYTHON=$((END_PYTHON - START_PYTHON))
    RUNTIME_C=$((END_C - START_C))

    echo "$ITERA,$RUNTIME_PYTHON,$RUNTIME_C" >> $RESULT

done
```

8.1.4. Generar gráficas pruebas

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sys
import csv
import matplotlib.pyplot as plt
import matplotlib
import numpy as np

log = sys.argv[1]

with open(log, 'rb') as csvfile:
    results = csv.reader(csvfile)

    x = []
    graph1 = []
    graph2 = []

    for row in results:
        x.append(int(row[0]))
        graph1.append(int(row[1]))
        graph2.append(int(row[2]))

    plt.style.use('fivethirtyeight')

    plt.plot(x, graph1, label='Python')
    plt.plot(x, graph2, label='C')

    plt.xlabel('Iteraciones')
    plt.ylabel('Tiempo de ejecucion (ms)')

    plt.legend()

    ax = plt.gca()
    ax.get_xaxis().get_major_formatter().set_scientific(False)


    fig = plt.gcf()
    fig.set_size_inches(18.5, 10.5)

    plt.savefig('../graphics/grafica_c_vs_python.pdf', transparent=True, dpi=1024
```

8.1.5. Segunda prueba simple en C

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

float f_simple(float x) {
    return pow((x-1), 8);
}

float f_prima_simple(float x) {
    return 8 * pow((x-1), 7);
}

int main(int argc, char ** args) {
    int i;
    float x0, x1;

    x1 = 0;
    i = 0;

    do {
        x0 = x1;
        x1 = x0 - f_simple(x0) / f_prima_simple(x0);

        i++;
    } while (x0 - x1 != 0);

    printf("%d %.30f\n", i, x1);
}
```

8.1.6. Segunda prueba doble en C

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

double f_double(double x) {
    return pow((x-1), 8);
}

double f_prima_double(double x) {
    return 8 * pow((x-1), 7);
}

int main(int argc, char ** args) {
    int i;
    double x0, x1;

    x1 = 0;
    i = 0;

    do {
        x0 = x1;
        x1 = x0 - f_double(x0) / f_prima_double(x0);

        i++;
    } while (x0 - x1 != 0);

    printf("%d %.30f\n", i, x1);
}
```

8.1.7. Segunda prueba mixta en C

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

float f_simple(float x) {
    return pow((x-1), 8);
}

float f_prima_simple(float x) {
    return 8 * pow((x-1), 7);
}

double f_double(double x) {
    return pow((x-1), 8);
}

double f_prima_double(double x) {
    return 8 * pow((x-1), 7);
}

int main(int argc, char ** args) {
    int i, j;
    float x0, x1;
    double x0d, x1d;

    x1 = 0;
    i = 0;

    do {
        x0 = x1;
        x1 = x0 - f_simple(x0) / f_prima_simple(x0);

        i++;
    } while (x0 - x1 != 0);

    x1d = x1;
    j   = 0;
    do {
        x0d = x1d;
        x1d = x0d - f_double(x0d) / f_prima_double(x0d);

        j++;
    } while (x0d - x1d != 0);

    printf("%d %d %d %.30f\n", i, j, i+j, x1d);
}
```

8.1.8. Makefile

```
default: main

main:
        gcc -lm prueba1.c -o prueba1 -Wall -O1
        gcc -lm prueba2_simple.c -o prueba2_simple -Wall -O1
        gcc -lm prueba2_doble.c -o prueba2_doble -Wall -O1
        gcc -lm prueba2_mixta.c -o prueba2_mixta -Wall -O1
```

8.2. Resolución numérica de ecuaciones diferenciales difusas

8.2.1. Método de Euler: Python

2.7-basic

```
#!/usr/bin/python
import numpy
import math
import sys

# Configuration
FUZZY_PARTITIONS_SIZE = int(sys.argv[1])
EULER_STEP_SIZE       = int(sys.argv[2])

# EDO definitions
def function_problem(x, y):
    return 2*x - 3*y + 1

def function_solution(x, ai):
    return (math.exp(-3*x) * (-1 + ai * 9 + math.exp(3*x) * (1 + 6*x))) / 9

# Variables
fuzzy_partitions      = numpy.linspace(-1, 1, FUZZY_PARTITIONS_SIZE)
x_domain              = numpy.linspace(0, 1, EULER_STEP_SIZE)

for ai in fuzzy_partitions:
    yi0 = ai

    for xj in x_domain:
        yij = yi0 + 1.0/EULER_STEP_SIZE * function_problem(xj, yi0)
        yi0 = yij

        # print("Error={}".format(abs(function_solution(xj, ai) - yi0)))
```

2.7-cython-basic

```
#!/usr/bin/python
import pyximport; pyximport.install()
import numpy
import math
import sys

# Configuration
FUZZY_PARTITIONS_SIZE = int(sys.argv[1])
EULER_STEP_SIZE       = int(sys.argv[2])

# EDO definitions
def function_problem(x, y):
    return 2*x - 3*y + 1

def function_solution(x, ai):
    return (math.exp(-3*x) * (-1 + ai * 9 + math.exp(3*x) * (1 + 6*x))) / 9

# Variables
fuzzy_partitions      = numpy.linspace(-1, 1, FUZZY_PARTITIONS_SIZE)
x_domain              = numpy.linspace(0, 1, EULER_STEP_SIZE)


for ai in fuzzy_partitions:
    yi0 = ai

    for xj in x_domain:
        yij = yi0 + 1.0/EULER_STEP_SIZE * function_problem(xj, yi0)
        yi0 = yij

        # print("Error={}".format(abs(function_solution(xj, ai) - yi0)))
```

2.7-cython-pool

```
#!/usr/bin/python
import pyximport; pyximport.install()
import numpy
import math
import sys

# Configuration
FUZZY_PARTITIONS_SIZE = int(sys.argv[1])
EULER_STEP_SIZE       = int(sys.argv[2])

# EDO definitions
def function_problem(x, y):
    return 2*x - 3*y + 1

def function_solution(x, ai):
    return (math.exp(-3*x) * (-1 + ai * 9 + math.exp(3*x) * (1 + 6*x))) / 9

# Variables
fuzzy_partitions      = numpy.linspace(-1, 1, FUZZY_PARTITIONS_SIZE)
x_domain              = numpy.linspace(0, 1, EULER_STEP_SIZE)


for ai in fuzzy_partitions:
    yi0 = ai

    for xj in x_domain:
        yij = yi0 + 1.0/EULER_STEP_SIZE * function_problem(xj, yi0)
        yi0 = yij

        # print("Error={}".format(abs(function_solution(xj, ai) - yi0)))
```

2.7-pool

```
#!/usr/bin/python
import numpy
import math
from multiprocessing import Pool
import sys

# Configuration
FUZZY_PARTITIONS_SIZE = int(sys.argv[1])
EULER_STEP_SIZE       = int(sys.argv[2])

# EDO definitions
def function_problem(x, y):
    return 2*x - 3*y + 1

def function_solution(x, ai):
    return (math.exp(-3*x) * (-1 + ai * 9 + math.exp(3*x) * (1 + 6*x))) / 9

# Variables
fuzzy_partitions      = numpy.linspace(-1, 1, FUZZY_PARTITIONS_SIZE)
x_domain              = numpy.linspace(0, 1, EULER_STEP_SIZE)

# Parallel task
def worker(ai):
    yi0 = ai

    for xj in x_domain:
        yij = yi0 + 1.0/EULER_STEP_SIZE * function_problem(xj, yi0)
        yi0 = yij

        # print("Error={}".format(abs(function_solution(xj, ai) - yi0)))

p = Pool(5)
p.map(worker, fuzzy_partitions)
```

3.5-basic

```
#!/usr/bin/python3
import numpy
import math
import sys

# Configuration
FUZZY_PARTITIONS_SIZE = int(sys.argv[1])
EULER_STEP_SIZE       = int(sys.argv[2])

# EDO definitions
def function_problem(x, y):
    return 2*x - 3*y + 1

def function_solution(x, ai):
    return (math.exp(-3*x) * (-1 + ai * 9 + math.exp(3*x) * (1 + 6*x))) / 9

# Variables
fuzzy_partitions      = numpy.linspace(-1, 1, FUZZY_PARTITIONS_SIZE)
x_domain              = numpy.linspace(0, 1, EULER_STEP_SIZE)

for ai in fuzzy_partitions:
    yi0 = ai

    for xj in x_domain:
        yij = yi0 + 1.0/EULER_STEP_SIZE * function_problem(xj, yi0)
        yi0 = yij

        # print("Error={}".format(abs(function_solution(xj, ai) - yi0)))
```

3.5-pool

```
#!/usr/bin/python
import numpy
import math
from multiprocessing import Pool
import sys

# Configuration
FUZZY_PARTITIONS_SIZE = int(sys.argv[1])
EULER_STEP_SIZE       = int(sys.argv[2])

# EDO definitions
def function_problem(x, y):
    return 2*x - 3*y + 1

def function_solution(x, ai):
    return (math.exp(-3*x) * (-1 + ai * 9 + math.exp(3*x) * (1 + 6*x))) / 9

# Variables
fuzzy_partitions      = numpy.linspace(-1, 1, FUZZY_PARTITIONS_SIZE)
x_domain              = numpy.linspace(0, 1, EULER_STEP_SIZE)

# Paralell task
def worker(ai):
    yi0 = ai

    for xj in x_domain:
        yij = yi0 + 1.0/EULER_STEP_SIZE * function_problem(xj, yi0)
        yi0 = yij

        # print("Error={}".format(abs(function_solution(xj, ai) - yi0)))

p = Pool(5)
p.map(worker, fuzzy_partitions)
```

Hacer test

```
#!/usr/bin/python
import os
from subprocess import call
import time
import json

# Configuration
RUN='main.py'

# Variables
results = {}


for folder in os.listdir('.'):
    if os.path.isdir(folder):
        script  = folder + "/" + RUN
        results[script] = []


        for i in range(0, 10000):
            print("[{}] Running test to: ".format(i) + script)

            start_time = time.time()
            call([script, str(i), '100'])
            results[script].append(time.time() - start_time)


output = json.dumps(results)

newfile = open("results.json", 'w')
newfile.write(output)
newfile.close()
```

Pintar test

```
#!/usr/bin/python
import json
import matplotlib.pyplot as plt
import numpy as np

# Read results JSON.
results_file = open('results.json', 'r')
results      = json.loads(results_file.read())
results_file.close()

plt.style.use('fivethirtyeight')

def reduce(l2, points):
    l      = list(l2)
    output = []
    step   = len(l) / points

    for i in range(0, points):
        output.append(l[step * i])
    return output

for test in results.keys():
    x    = range(0, 10000)
    y    = results[test]
    fit = np.polyfit(x,y,2)
    fit_fn = np.poly1d(fit)

    plt.plot(reduce(x, 2), reduce(fit_fn(x), 2), '-', label=test)

    print "El test {} ha tardado {} segundos".format(test, sum(y))


plt.xlabel('Particion')
plt.ylabel('Tiempo')
plt.legend(loc='upper left')


fig = plt.gcf()
fig.set_size_inches(18.5, 10.5)

plt.savefig('../../../graphics/grafica_python_euler_comparativa.pdf', transpa
```

8.2.2. Método de Euler: C

```c
#include <stdio.h>
#include <stdlib.h>

#include "math.h"

/**
 * For testing speed with different preccision.
 */
#ifdef DOUBLE_PRECCISION
    typedef double num;
#else
    typedef float  num;
#endif

/**
 * EDO to solve.
 */
num function_problem(num x, num y) {
    return 2*x - 3*y + 1;
}

/**
 * EDO solution.
 */
num function_solution(num x, num ai) {
    return (exp(-3*x) * (-1 + ai * 9 + exp(3*x) * (1 + 6*x))) / 9;
}

int main(int argc, char const *argv[])
{
    unsigned int fuzzy_partition_size, euler_step_size, i, j;
    num *fuzzy_partitions, *x_domain, yi0, yij, max_error;

    // Configuration
    fuzzy_partition_size = atoi(argv[1]);
    euler_step_size      = atoi(argv[2]);

    // Declare problem variables.
    fuzzy_partitions     = malloc(sizeof(num) * fuzzy_partition_size);
    x_domain             = malloc(sizeof(num) * euler_step_size);

    for (i = 0; i  < fuzzy_partition_size; i++) {
        fuzzy_partitions[i] = -1 + 2.0*i / (fuzzy_partition_size - 1);
    }

    for (i = 0;i < euler_step_size; i++) {
        x_domain[i] = 1.0 * i / (euler_step_size - 1);
```

```
    }

    max_error = 0;

    // Solve it.
    for (i = 0; i < fuzzy_partition_size; i++) {
        yi0 = fuzzy_partitions[i];

        for (j = 0; j < euler_step_size; j++) {
            yij = yi0 + 1.0/euler_step_size * function_problem(x_domain[j], y
            yi0 = yij;
        }
    }

    printf("Precission: %d\n", sizeof(num));

    return 0;
}
```

Makefile

```
all:
        gcc -o basic-simple main.c -lm
        gcc -o fastmath-simple main.c -lm -Ofast

        gcc -o basic-double main.c -lm -D DOUBLE_PRECCISION
        gcc -o fastmath-double main.c -lm -Ofast -D DOUBLE_PRECCISION
        gcc -o o3-double main.c -lm -O3 -D DOUBLE_PRECCISION
```

Hacer tests

```
#!/usr/bin/python
import os
from subprocess import call
import time
import json

# Variables
results = {}
for executable in ['./fastmath-simple', './fastmath-double']:
    if executable == 'main.c' or executable == 'Makefile':
        continue

    results[executable] = []

    for i in range(0, 10000):
        print("[{}] Running test to: ".format(i*10000) + executable)

        start_time = time.time()
        call([executable, str(10000*i), str(i*10000)])
        results[executable].append(time.time() - start_time)


output = json.dumps(results)

newfile = open("results2.json", 'w')
newfile.write(output)
newfile.close()
```

Pintar tests

```
#!/usr/bin/python
import json
import matplotlib.pyplot as plt
import numpy as np

# Read results JSON.
results_file = open('results.json', 'r')
results      = json.loads(results_file.read())
results_file.close()

plt.style.use('fivethirtyeight')

def reduce(l2, points):
    l      = list(l2)
    output = []
    step   = len(l) / points

    for i in range(0, points):
        output.append(l[step * i])
    return output

for test in results.keys():
    x   = np.linspace(0, 10000**2, 10000)
    y   = results[test]
    fit = np.polyfit(x,y,2)
    fit_fn = np.poly1d(fit)

    plt.plot(reduce(x, 2), reduce(fit_fn(x), 2), '-', label=test)

    print "El test {} ha tardado {} segundos".format(test, sum(y))



plt.xlabel('Particion')
plt.ylabel('Tiempo')
plt.legend(loc='upper left')


fig = plt.gcf()
fig.set_size_inches(18.5, 10.5)

plt.savefig('../../../graphics/grafica_cseq_euler_comparativa.pdf', transparent=T
```

8.2.3. Método de Runge-Kutta: C

```c
#include <stdio.h>
#include <stdlib.h>

#include "math.h"

/**
 * For testing speed with different preccision.
 */
#ifdef DOUBLE_PRECCISION
    typedef double num;
#else
    typedef float  num;
#endif

/**
 * EDO to solve.
 */
num function_problem(num x, num y) {
    return 2*x - 3*y + 1;
}

/**
 * EDO solution.
 */
num function_solution(num x, num ai) {
    return (exp(-3*x) * (-1 + ai * 9 + exp(3*x) * (1 + 6*x))) / 9;
}

int main(int argc, char const *argv[])
{
    unsigned int fuzzy_partition_size, euler_step_size, i, j;
    num *fuzzy_partitions, *x_domain, yi0, yij, max_error, h, k1, k2, k3, k4;

    // Configuration
    fuzzy_partition_size = atoi(argv[1]);
    euler_step_size      = atoi(argv[2]);

    // Declare problem variables.
    fuzzy_partitions     = malloc(sizeof(num) * fuzzy_partition_size);
    x_domain             = malloc(sizeof(num) * euler_step_size);

    for (i = 0; i  < fuzzy_partition_size; i++) {
        fuzzy_partitions[i] = -1 + 2.0*i / (fuzzy_partition_size - 1);
    }

    for (i = 0;i < euler_step_size; i++) {
        x_domain[i] = 1.0 * i / (euler_step_size - 1);
```

```
    }

    h = 1.0 / (euler_step_size - 1);
    max_error = 0;

    // Solve it.
    for (i = 0; i < fuzzy_partition_size; i++) {
        yi0 = fuzzy_partitions[i];

        for (j = 0; j < euler_step_size; j++) {
            k1  = function_problem(x_domain[i], yi0);
            k2  = function_problem(x_domain[i] + h/2, yi0 + 0.5 * k1 * h);
            k3  = function_problem(x_domain[i] + h/2, yi0 + 0.5 * k2 * h);
            k4  = function_problem(x_domain[i] + h, yi0 + k3 * h);

            yij = yi0 + 1.0 / 6 * h * (k1 + k2 + k3 + k4);
            yi0 = yij;
        }
    }

    printf("Precission: %d\n", sizeof(num));

    return 0;
}
```

Makefile

```
all:
	gcc -o basic-simple main.c -lm
	gcc -o fastmath-simple main.c -lm -Ofast

	gcc -o basic-double main.c -lm -D DOUBLE_PRECCISION
	gcc -o fastmath-double main.c -lm -Ofast -D DOUBLE_PRECCISION
	gcc -o o3-double main.c -lm -O3 -D DOUBLE_PRECCISION
```

Hacer tests

```
#!/usr/bin/python
import os
from subprocess import call
import time
import json

# Variables
results = {}
for executable in ['./fastmath-simple', './fastmath-double', './basic-simple', './

    results[executable] = []

    for i in range(0, 10000):
        print("[{}] Running test to: ".format(i) + executable)

        start_time = time.time()
        call([executable, '100', str(i)])
        results[executable].append(time.time() - start_time)


output = json.dumps(results)

newfile = open("results.json", 'w')
newfile.write(output)
newfile.close()
```

Pintar tests

```
#!/usr/bin/python
import json
import matplotlib.pyplot as plt
import numpy as np

# Read results JSON.
results_file = open('results.json', 'r')
results      = json.loads(results_file.read())
results_file.close()

plt.style.use('fivethirtyeight')

def reduce(l2, points):
    l      = list(l2)
    output = []
    step   = len(l) / points

    for i in range(0, points):
        output.append(l[step * i])
    return output

for test in results.keys():
    x   = np.linspace(0, 10000**2, 10000)
    y   = results[test]
    fit = np.polyfit(x,y,2)
    fit_fn = np.poly1d(fit)

    plt.plot(reduce(x, 2), reduce(fit_fn(x), 2), '-', label=test)

    print "El test {} ha tardado {} segundos".format(test, sum(y))



plt.xlabel('Particion')
plt.ylabel('Tiempo')
plt.legend(loc='upper left')


fig = plt.gcf()
fig.set_size_inches(18.5, 10.5)

plt.savefig('../../../graphics/grafica_cseq_rk_comparativa.pdf', transparent=
```

8.2.4. Método de Runge-Kutta: CUDA

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include "../../include/edo_fuzzy_solver/solver.cuh"
#include "../../include/utils/cuda.cuh"

__device__ double edo_fuzzy_solver_device_function_call(double x, double y) {
    return exp(-y * y);
}

__device__ unsigned long edo_fuzzy_solver_calculate_pointer(long i, long j, long
    return i * (n+1) + j;
}

edo_fuzzy_solver_error edo_fuzzy_solver_create(double * fuzzy_set, unsigned int f
    edo_fuzzy_solver_error error_t;

    error_t                  = EFS_E_OK;

    solver->fuzzy_set        = fuzzy_set;
    solver->fuzzy_set_length = fuzzy_set_length;
    solver->tol              = tol;

    return error_t;
}


__global__ void edo_fuzzy_solver_kernel_runge_kutta(double * fuzzy_set, double to
    unsigned long i, j, pointer;
    double k1, k2, k3, k4, x, y;

    i = threadIdx.x + blockIdx.x * blockDim.x;

    if (i >= fuzzy_set_length) {
        return;
    }

    // Initial value.

    pointer           = edo_fuzzy_solver_calculate_pointer(i, 0, fuzzy_set_length
    y_values[pointer] = fuzzy_set[i];

    for (j = 0; j < n; j++) {
```

```
            pointer           = edo_fuzzy_solver_calculate_pointer(i, j, fuzzy_se

            x_values[pointer] = j * tol;

            x                 = x_values[pointer];
            y                 = y_values[pointer];
            k1                = edo_fuzzy_solver_device_function_call(x, y);

            x                 = x_values[pointer] + .5 * tol;
            y                 = y_values[pointer] + .5 * tol * k1;
            k2                = edo_fuzzy_solver_device_function_call(x, y);

            x                 = x_values[pointer] + .5 * tol;
            y                 = y_values[pointer] + .5 * tol * k2;
            k3                = edo_fuzzy_solver_device_function_call(x, y);

            x                 = x_values[pointer] + tol;
            y                 = y_values[pointer] + tol * k3;
            k4                = edo_fuzzy_solver_device_function_call(x, y);

            y_values[pointer + 1] = y_values[pointer] + tol/6 * (k1 + 2*k2 + 2*k3
        }
    }

edo_fuzzy_solver_error edo_fuzzy_solver_solve(edo_fuzzy_solver solver, edo_fu
    int gridSize, blockSize, minGridSize;
    unsigned long n, i, j;
    edo_fuzzy_solver_error error_t;
    cudaError_t cuda_error;
    double* fuzzy_set_cuda, *x_values_cuda, *y_values_cuda, *x_values_raw, *y

    error_t = EFS_E_OK;
    n = 1.0/solver.tol;

    // Starts cuda.
    cuda_error = cudaSetDevice(0);

    if (cuda_error != cudaSuccess) {
        error_t = EFS_E_CUDA_ERROR;
        goto clean;
    }

    // Allocate cuda memory.
    cuda_error = cudaMalloc(&fuzzy_set_cuda, sizeof(double) * solver.fuzzy_se
    if (cuda_error != cudaSuccess) {
        error_t = EFS_E_CUDA_ERROR;
        goto clean;
    }
```

```
// Copy host memory to GPU.
cuda_error = cudaMemcpy(fuzzy_set_cuda, solver.fuzzy_set, solver.fuzzy_set_le
if (cuda_error != cudaSuccess) {
    error_t = EFS_E_CUDA_ERROR;
    goto clean;
}

cuda_error = cudaMalloc(&x_values_cuda, sizeof(double) * solver.fuzzy_set_leng
if (cuda_error != cudaSuccess) {
    error_t = EFS_E_CUDA_ERROR;
    goto clean;
}

cuda_error = cudaMalloc(&y_values_cuda, sizeof(double) * solver.fuzzy_set_leng
if (cuda_error != cudaSuccess) {
    error_t = EFS_E_CUDA_ERROR;
    goto clean;
}

cudaOccupancyMaxPotentialBlockSize(&minGridSize, &blockSize, edo_fuzzy_solver_
gridSize = (solver.fuzzy_set_length + blockSize - 1) / blockSize;

// Run runge kutta in kernel.
edo_fuzzy_solver_kernel_runge_kutta << <gridSize, blockSize >> >(fuzzy_set_cud

cuda_error = cudaDeviceSynchronize();
if (cuda_error != cudaSuccess) {
    error_t = EFS_E_CUDA_ERROR;
    goto clean;
}

// Now pass the result to CPU.
x_values_raw = (double*)malloc(sizeof(double) * solver.fuzzy_set_length * n);
y_values_raw = (double*)malloc(sizeof(double) * solver.fuzzy_set_length * n);

cuda_error = cudaMemcpy(x_values_raw, x_values_cuda, solver.fuzzy_set_length *
if (cuda_error != cudaSuccess) {
    error_t = EFS_E_CUDA_ERROR;
    goto clean;
}

cuda_error = cudaMemcpy(y_values_raw, y_values_cuda, solver.fuzzy_set_length *
if (cuda_error != cudaSuccess) {
    error_t = EFS_E_CUDA_ERROR;
    goto clean;
}

solution->x_values = (double**)realloc(solution->x_values, sizeof(double*) *
solution->y_values = (double**)realloc(solution->y_values, sizeof(double*) *
```

```
    for (i = 0; i < solver.fuzzy_set_length;i++) {
        solution->x_values[i] = (double*)malloc(sizeof(double) * n);
        solution->y_values[i] = (double*)malloc(sizeof(double) * n);

        for (j = 0; j < n; j++) {
            solution->x_values[i][j] = x_values_raw[i * (solver.fuzzy_set_len
            solution->y_values[i][j] = y_values_raw[i * (solver.fuzzy_set_len
        }
    }

    solution->points = n;
 clean:
    if (!fuzzy_set_cuda) {
        cudaFree(fuzzy_set_cuda);
    }

    if (!x_values_cuda) {
        cudaFree(x_values_cuda);
    }

    if (!x_values_cuda) {
        cudaFree(x_values_cuda);
    }

    if (!x_values_raw) {
        free(x_values_raw);
    }

    if (!y_values_raw) {
        free(y_values_raw);
    }

    if (cuda_error != cudaSuccess) {
        fprintf(stderr,"CUDA assert: %s\n", cudaGetErrorString(cuda_error));
    }
    return error_t;
}

double edo_fuzzy_solver_evaluate(double x, unsigned int fuzzy_value, edo_fuzz
    return NAN;
}
```

BIBLIOGRAFÍA

[1] Luciana Takata Gomes, Laécio Carvalho de Barros, and Barnabas Bede. Fuzzy differential equations in various approaches. *Maths*, 1:11–38, 2015.

[2] Scholarpedia. Fuzzy sets - http://www.scholarpedia.org/article/fuzzy_sets.

[3] Jesus Medina Moreno and Elena Medina Reus. Apuntes de modelización matemática. *Maths*, 3:1–20, 2018.

[4] Samuel Corveleyn. The numerical solution of elliptic partial differential equations with fuzzy coefficients. 06 2014.

[5] M. Puri and D. Ralescu. Fuzzy random variables. *Maths*, 1:409–422, 1986.

[6] M. Fukuhara. *Intégration des applications mesurables dont la valeur est un compact convexe*. Publications of the Research Institute for Mathematical Sciences. Kyoto University. Series B. Inst., Univ., 1967.

[7] B. Bede. *Mathematics of Fuzzy Sets and Fuzzy Logic*. 2013.

[8] Osmo Kaleva. The cauchy problem for fuzzy differential equations. fuzzy sets syst. 35, 389-396. *Fuzzy Sets and Systems*, 35:389–396, 05 1990.

[9] Çagin Ararat and Birgit Rudloff. A characterization theorem for aumann integrals. *Set-valued and Variational Analysis*, 23(2):305–318, 2015.

[10] Madan L Puri and Dan A Ralescu. Fuzzy random variables. *Journal of Mathematical Analysis and Applications*, 114(2):409 – 422, 1986.

[11] O. Kaleva. Fuzzy differential equations. 1987.

[12] Barnabás Bede and Luciano Stefanini. Generalized differentiability of fuzzy-valued functions. *Fuzzy Sets and Systems*, 230:119 – 141, 2013. Differential Equations Over Fuzzy Spaces - Theory, Applications, and Algorithms.

[13] Y. Chalco-Cano and H. Román-Flores. Comparation between some approaches to solve fuzzy differential equations. *Fuzzy Sets Syst.*, 160(11):1517–1527, June 2009.

[14] Y. Chalco-Cano and H. Román-Flores. Comparation between some approaches to solve fuzzy differential equations. *Fuzzy Sets Syst.*, 160(11), June 2009.

[15] E. Hüllermeier. An approach to modeling and simulation of uncertain dynamical systems. 1997.

[16] P. Diamond. Time-dependent differential inclusions, cocycle attractors and fuzzy differential equations.

[17] P. Diamond. Brief note on the variation of constants formula for fuzzy differential equations.

[18] José Carlos García Ortega. Breve introducción a la programación en paralelo - https://github.com/josecarlosgarcia95/presentacionprogramacionparalelo, 2018.

[19] José Carlos García Ortega. Experimentos numéricos - https://github.com/josecarlosgarcia95/experimentosnumericos, 2018.

[20] Linus Torvalds and dewar. What is acceptable for -ffast-math? (was: associative law incombine). 2001.

[21] NASA. ASTRONOMICAL CONSTANTS - http://asa.usno.navy.mil/static/files/2016/astronomical_constants_2016.pd

[22] Alireza K Golmankhaneh and Ahmad Jafarian. About fuzzy schrödinger equation. *2014 International Conference on Fractional Differentiation and Its Applications, ICFDA 2014*, pages 1–5, 06 2014.

[23] Amir Sadeghi, Ahmad Izani Md. Ismail, and Ali Jameel. Solving systems of fuzzy differential equation. *International Mathematical Forum*, 6:2087 – 2100, 09 2011.

[24] Kausik Kumar Majumdar and D. Dutta Majumder. Fuzzy differential inclusions in atmospheric and medical cybernetics. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34:877–887, 2004.

[25] T.H. Skopinski, K.G. Johnson, and Langley Research Center. *Determination of azimuth angle at burnout for placing a satellite over a selected earth position*. NASA technical note. National Aeronautics and Space Administration, 1960.

[26] Shadan Sadigh Behzadi. Solving cauchy reaction-diffusion equation by using picard method. 2013.

[27] Douglas C. Giancoli. *Physics for Scientists and Engineers*. Pearson, 2014.

[28] Jesús Rojo. Métodos matemáticos i. *Universidad de Valladolid*, 2012-2013.