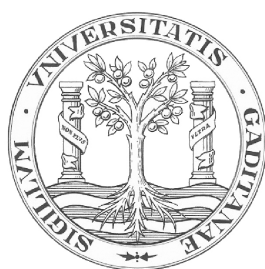


UNIVERSIDAD DE CÁDIZ

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

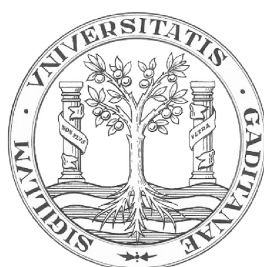
Dr. Jesús Medina Moreno

Este documento esta realizado bajo licencia [Creative Commons](#) "Reconocimiento-CompartirIgual 3.0 España".



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, diciembre 2018

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 ordinaria, 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 se introducen algunos modelos que se pueden construir teniendo en cuenta la naturaleza difusa de la vida real.

Finalmente, en este trabajo también se trabajan técnicas computacionales de alto rendimiento, para conseguir el mejor rendimiento energético y en cuestión de tiempo. Y se verá, que las técnicas computacionales de alto rendimiento pueden cambiar de forma bastante notable los resultados de las pruebas.

Agradecimientos

Este trabajo no podría haber sido posible sin las personas que han ido recorriendo 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 a día de hoy 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 de su carácter luchador.

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.

1. Estado del arte	1
1.1. Marco teórico	1
1.2. Objetivos de este trabajo	2
2. Conjuntos difusos	5
2.1. Subconjuntos difusos	5
2.1.1. Subconjuntos difusos	6
2.1.2. α -corte	6
2.2. Números difusos	7
2.2.1. Caracterización números difusos	7
2.3. Principio de extensión de Zadeh	8
2.3.1. Teoremas de continuidad	9
2.4. Aritmética difusa	11
2.4.1. Aritmética en conjuntos clásicos	11
2.4.2. Aritmética en conjuntos difusos	11
2.4.3. Hukuhara y diferencia generalizada	12
2.5. Interactividad	13
2.6. Métrica en conjuntos difusos	14
2.7. Funciones difusas	15
2.7.1. Continuidad de funciones difusas	15
3. Ecuaciones diferenciales difusas	17
3.1. Cálculo difuso para funciones definidas en conjuntos difusos	17
3.1.1. Distintas definiciones de derivada	17
3.1.2. Integral	20
3.1.3. Teorema fundamental del cálculo	22
3.2. Ecuaciones diferenciales difusas	22
3.2.1. Introducción	23

3.2.2.	Teorema de equivalencia entre EDO y EDD	23
3.2.3.	Inclusiones diferenciales	24
4.	Computación científica de alto rendimiento	27
4.1.	Conceptos básicos	27
4.1.1.	Memoria RAM	28
4.1.2.	Procesador (CPU)	28
4.1.3.	Tarjeta gráfica (GPU)	28
4.1.4.	Tarjeta de Red (Internet/Intranet/Cluster)	29
4.2.	Técnicas de alto rendimiento	29
4.2.1.	Programación de bajo nivel	29
4.2.2.	Precisión mixta	30
4.2.3.	Paralelización de algoritmos	32
4.2.4.	Optimizar compilación	33
5.	Modelos de ecuaciones diferenciales difusos	35
5.1.	Aplicaciones a las ciencias naturales	35
5.1.1.	Aplicación a la mecánica clásica	35
5.1.2.	Aplicación a la mecánica cuántica	37
6.	Resolución numérica de ecuaciones diferenciales difusas	39
6.1.	El método de Euler	39
6.1.1.	Método de Euler: Versión clásica	39
6.1.2.	Método de Euler: Versión difusa	40
6.1.3.	Método de Euler: Ejemplo	40
6.1.4.	Experimentos numéricos	41
6.2.	Método de Runge-Kutta de cuarto orden	45
6.2.1.	Método de Runge-Kutta de cuarto orden: Versión clásica	45
6.2.2.	Método de Runge-Kutta de cuarto orden: Versión difusa	46
6.2.3.	Método de Runge-Kutta de cuarto orden: Ejemplo	46
6.2.4.	Experimentos numéricos	46
6.3.	Comparativas	47
6.4.	Método difuso paralelizado	48
7.	Conclusiones	49
7.1.	Planes de futuro	50
8.	Apéndice: Código fuente	51
8.1.	Computación científica de alto rendimiento	51
8.1.1.	Prueba simple en Python	51
8.1.2.	Prueba simple en C	52
8.1.3.	Script de pruebas	53
8.1.4.	Generar gráficas pruebas	54
8.1.5.	Segunda prueba simple en C	55

8.1.6.	Segunda prueba doble en C	56
8.1.7.	Segunda prueba mixta en C	57
8.1.8.	Makefile	58
8.2.	Resolución numérica de ecuaciones diferenciales difusas	58
8.2.1.	Método de Euler: Python	58
8.2.2.	Método de Euler: C	66
8.2.3.	Método de Runge-Kutta: C	71
8.2.4.	Método de Runge-Kutta: CUDA	76

“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 prohíbe 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>

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

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 funciona utilizando la idea que subyace 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:

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

Por otro lado, 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}, μ_B) donde $\mu_B : \mathbb{U} \rightarrow [0, 1]$ y \mathbb{U} es el conjunto de discurso. ([Subconjunto difuso](#))

Dentro de la teoría de conjuntos difusos también se pueden desarrollar una serie de términos para tratar de definir lo que es un número difuso, se puede encontrar más información en el primer capítulo, pero básicamente cuando se habla de número difuso es un conjunto difuso que cumple que es un conjunto normal y convexo. Un ejemplo clásico de número difuso corresponde a los números triangulares:

1.2. OBJETIVOS DE ESTE TRABAJO

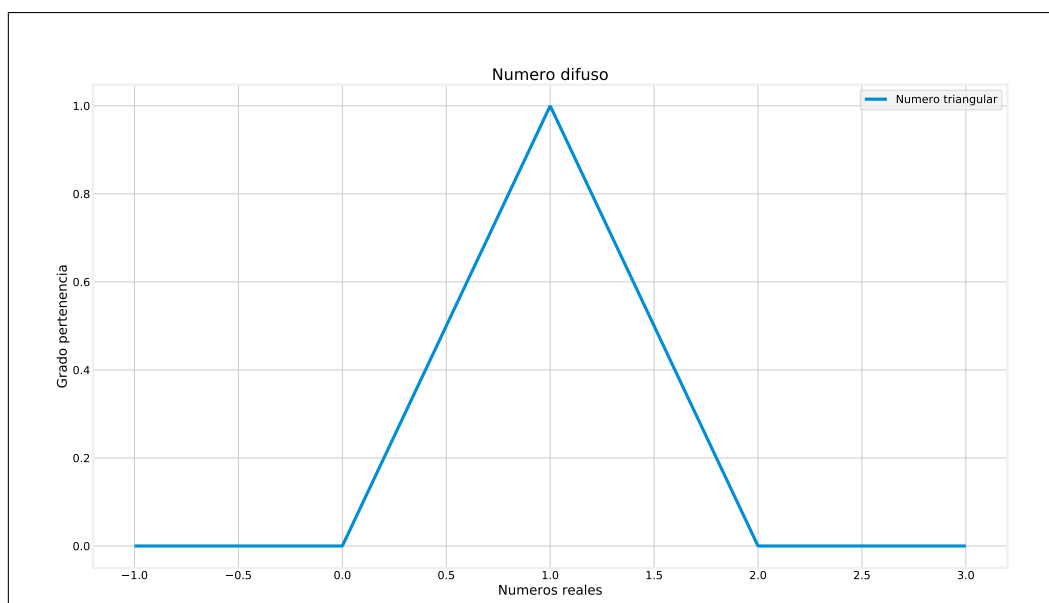


Figura 1.1: Ejemplo de número triangular

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 funcionan 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 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 herramientas difusas y técnicas computacionales avanzadas, lo que se quiere conseguir con este trabajo es:

- 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.

1. ESTADO DEL ARTE

- Comparar las soluciones de una ecuación diferencial difusa y una ecuación diferencial ordinaria.
- 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.

“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 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. SUBCONJUNTOS DIFUSOS

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}, μ_A) con:*

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

Se denomina a μ_A función de pertenencia.

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 cumpliera 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:*

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

Donde, cl define la clausura topológica del conjunto. Además, se define:

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

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

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]

2. CONJUNTOS DIFUSOS

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

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

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 difusos, 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 $\text{núcleo } A \neq \emptyset$.*

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

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

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.

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 Stacking). *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 $[A]_{\alpha_1} \subset [A]_{\alpha_2}$*

2.3. PRINCIPIO DE EXTENSIÓN DE ZADEH

3. Para toda sucesión no decreciente de $\alpha_n \in [0, 1]$ 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:

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

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

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 $\alpha_n \in [0, 1]$ 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 $U = \mathbb{R}$ y sea $\mu_A : \mathbb{R} \rightarrow [0, 1]$ definida de la siguiente forma:

$$\mu_A(x) = \begin{cases} \frac{x-a}{b-a} & \text{si } x \in [a, b] \\ \frac{c-x}{c-b} & \text{si } x \in (b, c] \\ 0, & \text{si } x \notin [a, c] \end{cases}$$

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

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.

2. CONJUNTOS DIFUSOS

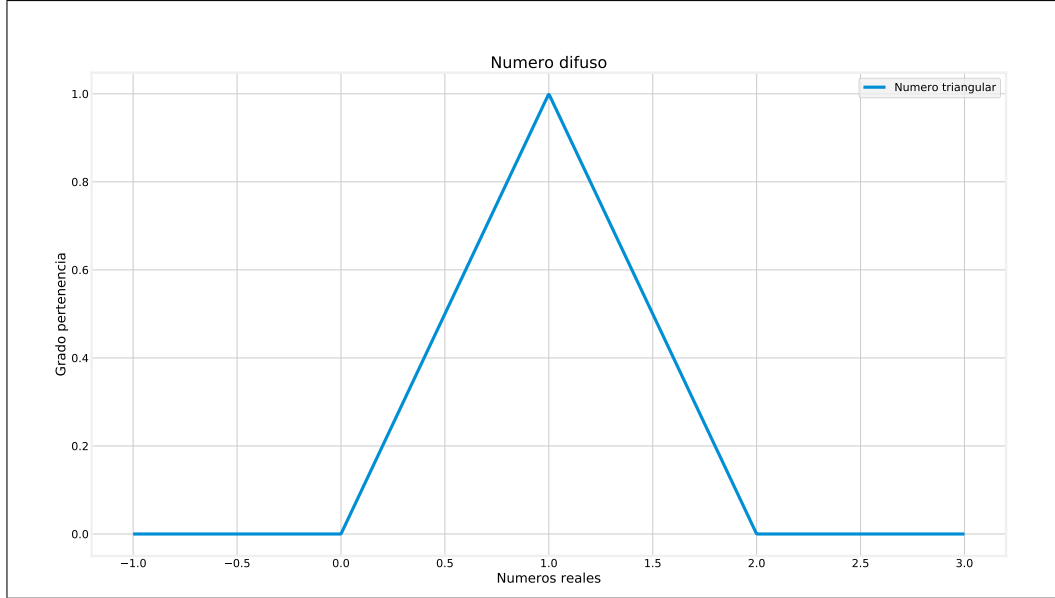


Figura 2.1: Ejemplo de número triangular

Definición 7 (Principio de extensión de Zadeh). Sean \mathbb{U} y \mathbb{V} dos conjuntos de 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

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

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

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

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. Se introduce en primer lugar la notación $\mathcal{F}_{\mathcal{H}}(\mathbb{U})$, que indica la familia de conjuntos de \mathbb{U} tales que sus α -cortes son no vacíos, compactos y convexos en \mathbb{U}

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 $y \in \mathbb{R}^m$

2.3. PRINCIPIO DE EXTENSIÓN DE ZADEH

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 Hausdorff, 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.

- $[\hat{f}(A)]_{\alpha} \subseteq f([A]_{\alpha})$. 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 $[\hat{f}(A)]_{\alpha} \supseteq f([A]_{\alpha})$. Dado que \mathbb{V} y \mathbb{U} son espacios de Hausdorff, 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 compactos, por tanto $f^{-1}(y) \cap [A]_0$ es compacto. Para $\alpha > 0$, sea $y \in [\hat{f}(A)]_{\alpha}$. Entonces $\mu_{\hat{f}(A)}(y) = \sup_{x \in f^{-1}(y)} \mu_A(x) \geq \alpha > 0$, y además, existe un $x \in f^{-1}(y)$ tal que $f^{-1}(y) \cap [A]_0 \neq \emptyset$.

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 tiene:

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

2. CONJUNTOS DIFUSOS

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 $\alpha \in [0, 1]$ \square

2.4. Aritmética difusa

El siguiente paso para poder construir métodos numéricos es necesario 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, se centrará en dar todas las operaciones en términos de α – cortes.

En primer lugar, se recordarán las definiciones habituales de las operaciones aritmética en 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 básicas de 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:

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

Y dado que las operaciones aritméticas son funciones continuas, es equivalente trabajar con los α – corte, aplicando el principio de extensión de Zadeh tenemos:

Sean A y B dos números difusos con α – corte dados por $[A]_\alpha = [a_\alpha^-, a_\alpha^+]$ y $[B]_\alpha = [b_\alpha^-, b_\alpha^+]$, se puede definir entonces las operaciones aritméticas como:

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

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

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

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

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:

$$\lim_{h \rightarrow 0^+} \frac{f(x+h) - f(x)}{h}$$

Se considera ahora una función $f(x) = A \in \mathcal{F}_{\mathcal{H}}(\mathbb{U})$, donde A es un número difuso constante.

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

Sean $[A]_\alpha$ los α -cortes de A entonces:

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

Si $A \neq 0$ se tiene que $[A - A]_\alpha \neq 0$, por tanto al dividir por $h \rightarrow 0$, tenemos una indeterminación, de donde, tal y como se ha definido las operaciones aritméticas para los conjuntos difusos no estarían definidas las derivadas de funciones constantes, y esto, es un problema. Necesitamos definir un nuevo concepto de diferencia, que para ello, se introduce 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, por tanto $[A]_\alpha = [a_\alpha^-, a_\alpha^+]$, $[B]_\alpha = [b_\alpha^-, b_\alpha^+]$ y $[C]_\alpha = [c_\alpha^-, c_\alpha^+]$.

De donde,

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

Por tanto,

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

2. CONJUNTOS DIFUSOS

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

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

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.

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: $\mu_{\hat{a}, \hat{b}} : V \times W \rightarrow [0, 1]$*

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 pertenencia conjunta). *Se define la función de pertenencia marginal respecto a como:*

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

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 $\mu_{\hat{a}, \hat{b}} = \min(\mu_{\hat{a}}(a), \mu_{\hat{b}}(b))$*

2.6. MÉTRICA EN CONJUNTOS DIFUSOS

Ejemplo 2. En el caso de conjuntos no interactivos, se escriben las operaciones aritméticas de los conjuntos difusos de la siguiente manera:

$$[\hat{a} + \hat{b}]_\alpha = [[\hat{a}_\alpha^- + [\hat{b}]_\alpha^-, [\hat{a}]_\alpha^+ + [\hat{b}]_\alpha^+]$$

$$[\hat{a} - \hat{b}]_\alpha = [[\hat{a}_\alpha^- - [\hat{b}]_\alpha^-, [\hat{a}]_\alpha^+ - [\hat{b}]_\alpha^+]$$

$$[\hat{a} * \hat{b}]_\alpha = \text{máx}\{[\hat{a}]_\alpha^i [\hat{b}]_\alpha^j\}, i, j \in \{+, -\}$$

$$[\hat{a}/\hat{b}]_\alpha = \text{máx}\{[\hat{a}]_\alpha^i / [\hat{b}]_\alpha^j\}, i, j \in \{+, -\}$$

2.6. Métrica en conjuntos difusos

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 de un espacio métrico \mathbb{U} compactos. 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 Hausdorff

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

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

Si A y B fueran números difusos, tendríamos:

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

Teorema 5 ([4]). El espacio de los números difusos, con la métrica d_∞ es un espacio de Banach.

2.7. Funciones difusas

En la literatura sobre funciones difusas existen dos definiciones para referirnos a funciones difusas, originalmente Dubois y Prade (1980) definieron los siguientes conceptos:

Definición 16 (Funciones definidas en conjuntos difusos). *Se dice que una función F es una función definida en un conjunto difuso si:*

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

Ejemplo 3. La función $f(x) = Ax$ donde $A = [a, b]$ con $a, b \in \mathbb{R}$, es una función definida en conjuntos difusos. Sus imágenes son intervalos.

Ejemplo 4. La función $f(x) = Ax$ donde $A = (a; b; c)$ con $a < b < c$, es una función definida 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 sobre conjuntos continua). *Sea $F : \Omega \rightarrow \mathcal{P}(\mathbb{R}^n)$, $\Omega \subset \mathbb{R}^m$ 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 $F : \Omega \rightarrow \mathcal{F}_{\mathcal{H}}(\mathbb{R}^n)$, $\Omega \subset \mathbb{R}^m$ 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 inferiormente 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.

“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 lo que está pasando.

3.1. Cálculo difuso para funciones definidas en conjuntos difusos

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 ([\[5\]](#))

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

$$\lim_{h \rightarrow 0^+} \frac{F(x_0+h) \ominus_H F(x_0)}{h} \quad \parallel \quad \lim_{h \rightarrow 0^+} \frac{F(x_0) \ominus_H F(x_0-h)}{h}$$

Existen, y son iguales a cierto elemento $F'_H(x_0) \in \mathcal{F}_H(\mathbb{R}^n)$, 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

3.1. CÁLCULO DIFUSO PARA FUNCIONES DEFINIDAS EN CONJUNTOS DIFUSOS

Ejemplo 5 (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 6 (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,

$$\lim_{h \rightarrow 0^+} \frac{(-h; 0; h)}{h} = A \parallel \lim_{h \rightarrow 0^+} \frac{(-h; 0; h)}{h} = A$$

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 [6]). 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 [7]). Sea $F : [a, b] \rightarrow \mathcal{F}_H(\mathbb{R}^n)$ una función H -Diferenciable, entonces F es continua en el sentido visto en el Tema 1.

Teorema 7 (Álgebra en derivadas [7]). Sean $F, G : [a, b] \rightarrow \mathcal{F}_H(\mathbb{R}^n)$ 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 $F : [a, b] \rightarrow \mathcal{F}_H(\mathbb{R})$ 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 [7]). Si $F : [a, b] \rightarrow \mathcal{F}_H(\mathbb{R}^n)$ es H-Diferenciable entonces $f_\alpha^-(x)$ y $f_\alpha^+(x)$ 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 6). Para evitar estos problemas, se introdujo el concepto de derivada fuertemente generalizada:

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

1.

$$\lim_{h \rightarrow 0^+} \frac{F(x_0+h) \ominus_H F(x_0)}{h} \parallel \lim_{h \rightarrow 0^+} \frac{F(x_0) \ominus_H F(x_0-h)}{h}$$

2.

$$\lim_{h \rightarrow 0^+} \frac{F(x_0+h) \ominus_H F(x_0)}{-h} \parallel \lim_{h \rightarrow 0^+} \frac{F(x_0) \ominus_H F(x_0-h)}{-h}$$

3.

$$\lim_{h \rightarrow 0^+} \frac{F(x_0+h) \ominus_H F(x_0)}{h} \parallel \lim_{h \rightarrow 0^+} \frac{F(x_0) \ominus_H F(x_0-h)}{-h}$$

4.

$$\lim_{h \rightarrow 0^+} \frac{F(x_0+h) \ominus_H F(x_0)}{-h} \parallel \lim_{h \rightarrow 0^+} \frac{F(x_0) \ominus_H F(x_0-h)}{h}$$

Existen y son iguales a algún elemento $F'_G(x_0)$ de $\mathcal{F}_H(\mathbb{R}^n)$, 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:

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

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 Hukuhara

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

$$\lim_{h \rightarrow 0} \frac{F(x_0 + h) \ominus_g F(x_0)}{h}$$

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

En resumen,

$$\text{H-Diferenciable} \Rightarrow \text{GH-Diferenciable} \Rightarrow \text{gH-Diferenciable} \Rightarrow \text{g-Diferenciable}$$

3.1.2. Integral

La primera propuesta de integral difusa es basada en el concepto integral de Aumann [8] para funciones multievaluadas. La definición original podemos verla en [9] y [10].

Por simplificar, vamos a denotar:

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

Con $G : I \rightarrow \mathcal{P}(\mathbb{R}^n)$

Integral de Aumann

Definición 24 (Integral de Aumann). La integral de Aumann de una función sobre un conjunto difuso $F : [a, b] \rightarrow \mathcal{F}_H(\mathbb{R}^n)$ sobre $[a, b]$ es definida 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 $(A) \int_a^b F(x) dx \in \mathcal{F}_H(\mathbb{R}^n)$

Integral de Riemann

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

Definición 25 (Integral de Riemann). La integral de Riemann de una función difusa $F : [a, b] \rightarrow \mathcal{F}_C(\mathbb{R})$ sobre $[a, b]$ es el número difuso A tal que para todo $\varepsilon > 0$ existe un $\delta > 0$ tal que para cualquier partición $\mathcal{P} := a = x_0 < x_1 < \dots < x_n = b$ con $x_i - x_{i-1} < \delta, i = 1, \dots, n$ 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 [10] que resumen en cierta manera algunas propiedades ya conocidas del cálculo infinitesimal clásico:

Teorema 9. Si una función $F : [a, b] \rightarrow \mathcal{F}_H(\mathbb{R})$ 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 $F : [a, b] \rightarrow \mathcal{F}_H(\mathbb{R})$ 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 $F, G : [a, b] \rightarrow \mathcal{F}_H(\mathbb{R})$ 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 [10]). Sea $F : [a, b] \rightarrow \mathcal{F}_H(\mathbb{R}^n)$ una función continua, entonces $G(x) = \int_a^x F(s)ds$ es H -Diferenciable y además,

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

Teorema 13 ([10]). Sea $F : [a, b] \rightarrow \mathcal{F}_H(\mathbb{R}^n)$ una función H -Diferenciable y sea F'_H su derivada integrable sobre $[a, b]$. Entonces,

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

Para todo $x \in [a, b]$

Hay un teorema parecido al anterior, pero esta vez exigiendo que la función es diferenciable fuertemente generalizada (2), que dice lo siguiente:

Teorema 14 ([11]). Sea $F : [a, b] \rightarrow \mathcal{F}_H(\mathbb{R}^n)$ una función diferenciable fuertemente generalizada (2) y sea F'_H su derivada 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 [12]

3.2.1. Introducción

En esta sección vamos a empezar planteando un problema difuso de la misma manera que se introducen las ecuaciones diferenciales clásicas:

$$X'(t) = f(t, X(t)), \quad X(0) = X_0 \quad (3.1)$$

Donde $f : [0, T] \times \mathcal{F}_{\mathcal{H}}(\mathbb{U}) \rightarrow \mathcal{F}(\mathbb{R}^n)$ se obtiene mediante el [principio de extensión de Zadeh](#) aplicada a una función continua $g : [0, T] \times U \rightarrow \mathbb{R}^n$. 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:

$$x'(t) = g(t, x(t)), \quad x(0) = c \quad (3.2)$$

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 [13]). *Sea U un conjunto abierto sobre \mathbb{R}^n y sea $[X_0]_{\alpha} \subset U$ 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 continua 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 7. *Se plantea la siguiente ecuación diferencial difusa con valores iniciales:*

$$\begin{cases} X' &= -X(t) \\ X(0) &= C \end{cases}$$

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;

$$\begin{cases} x' &= -x(t) \\ x(0) &= c \end{cases}$$

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 continua 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 8. Se plantea ahora un problema un tanto más interesante:

$$\begin{cases} X' &= X^2(t) \\ X &= C \end{cases}$$

Donde esta vez C es un número difuso triangular:

$$C(y) = \begin{cases} 3-y & \text{si } 2 \leq y \leq 3 \\ y-1 & \text{si } 1 \leq y \leq 2 \\ 0 & \text{en otro caso} \end{cases}$$

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

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

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 $\alpha \in [0, 1]$

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

3.2.3. Inclusiones diferenciales

Esta sección está basada en las observaciones que se pueden ver en los artículos [\[14\]](#), [\[15\]](#) y [\[16\]](#).

En estos artículos Hüllermeier y Diamond interpretan las [Ecuaciones Diferenciales Difusas](#) como una familia inclusiones diferenciales:

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

Bajo unas determinadas hipótesis, podemos afirmar:

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

Son los α -corte 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](#)

3. ECUACIONES DIFERENCIALES DIFUSAS

Teorema 16 ([13]). 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 continua en U para cada $t \in [0, T]$. Entonces, la solución difusa del problema 3.1 y las soluciones del problema de *Familia de inclusiones diferenciales* coinciden, es decir;

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

Para todo $t \in [0, T]$

“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 cálculo 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 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 trata más los problemas desde el punto de vista informático, que serán nuestras herramientas para desarrollar correctamente la resolución numérica de los problemas planteados.

Este capítulo está basado en los puntos que se exponen en [17].

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, pues todo el mundo ha hablado alguna vez de la memoria RAM de su ordenador, 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 desde 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 simultánea. 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 las tarjetas gráficas 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 también, cuando conectamos varios ordenadores y los coordinamos para realizar una tarea, se les llaman *cluster*.

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.

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 comporta Python y C, con el mismo código, en términos de tiempo de ejecución, uso de RAM: (Código del ejemplo):

- C es más eficiente a la hora de administrar la memoria RAM, al probar 10000000000 iteraciones, Python no puede alocar 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 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 C debe ser nuestra primera elección. (Ver figura [4.1](#))

4.2. TÉCNICAS DE ALTO RENDIMIENTO

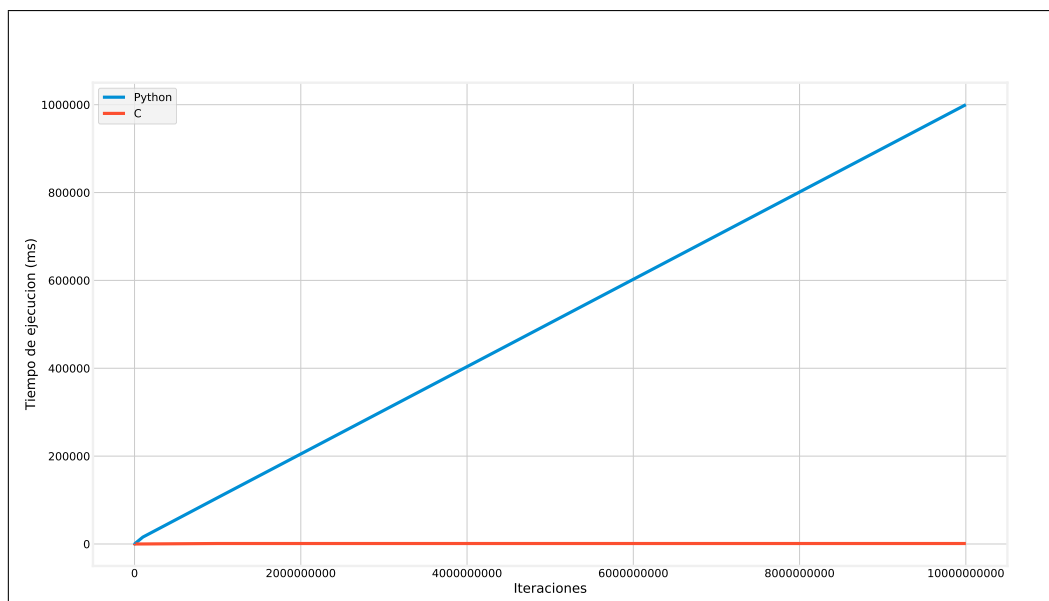


Figura 4.1: Gráfica comparativa C vs Python 8.1.1, 8.1.2

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 a 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 .
- **Precisión doble:** Los números se representan utilizando 8 bytes, por tanto, podemos representar 256^8 .

Generalmente, los procesadores están optimizados para trabajar mejor que con simple o 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 es el siguiente:

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

4. COMPUTACIÓN CIENTÍFICA DE ALTO RENDIMIENTO

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

Ejemplo 9 (Método de Newton precisión mixta, comparativa y desarrollo). *Supongamos que queremos encontrar las raíces 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 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: 0.999998152256011962890625000000*

Lo resolvemos ahora para doble precisión:

- *Tiempo de ejecución: 0m0,001s*
- *Iteraciones en simple precisión: 0*
- *Iteraciones en doble precisión: 265*
- *Resultado: 0.99999999999999555910790149937*

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: 0.99999999999999555910790149937*

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 n iteración no depende del resto de iteraciones.

Para conseguir la paralelización, podemos hacerlo mediante diferentes técnicas:

- 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 desempeño que los de la CPU, sin embargo, 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 10 (Diferencias finitas: Secuencial VS. Paralelo VS. GPU [18]). *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 y hemos utilizado OpenMP para generar una versión en paralelo de nuestro código inicial en C. Con esto conseguimos:

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

En la siguiente tabla, mostramos el tiempo de ejecución en segundos de cada uno de los programas utilizando las distintas técnicas:

4. COMPUTACIÓN CIENTÍFICA DE ALTO RENDIMIENTO

Mallado	C-No paralelo	CPU	GPU
5X5	0,001	0,001	0,441
25X25	0,001	0,001	0,441
50X50	0,001	0,001	0,441
70X70	0,001	0,001	0,441
100X100	0,001	0,001	0,484
1000X1000	0,005	0,005	0,472
10000X10000	1,178	0,624	0,997
20000X20000	8,961	7,754	2,594
22760X22760	22,99	9,227	3,221

A continuación, mostramos el consumo energético en $\mu A/s$:

Mallado	C-No paralelo	CPU	GPU
5X5	0,1	0,23	57,33
25X25	0,1	0,23	57,33
50X50	0,1	0,23	57,33
70X70	0,1	0,23	57,33
100X100	0,1	0,23	62,92
1000X1000	0,5	1,15	61,36
10000X10000	117,8	143,52	129,61
20000X20000	896,1	1783,42	337,22
22760X22760	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, se puede escribir la flag -Ofast a la hora de compilar nuestro programa. Se puede encontrar una discusión sobre esta flag entre desarrolladores de GCC y el propio Linus Torvalds, creador de Linux. [19]

El flag 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 reemplaza con las propiedades or-

4.2. TÉCNICAS DE ALTO RENDIMIENTO

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

En el último capítulo veremos una implementación de esto, aunque aquí se muestra un pequeño resumen de lo que supone la optimización en cuestión de tiempos:

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

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

Test	Consumo energético
Fastmath simple	118,644028 μ A/s
Fastmath doble	118,644058 μ A/s
Simple	152,542389 μ A/s
Doble	169,491638 μ 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

La construcción de los capítulos anteriores ha tenido como uno de sus objetivos principales la aplicación de las teorías expuestas a la realidad. Es por ello, que en este capítulo se analizará el potencial y la exactitud de estos métodos para modelar la realidad al añadir una serie de constantes difusas.

Se verá, como ejemplo, que sucede cuando se tiene en cuenta la incertidumbre estándar como números difusos.

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 [20] 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

5.1. APLICACIONES A LAS CIENCIAS NATURALES

Orbita planetaria

Ejemplo 11. 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

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

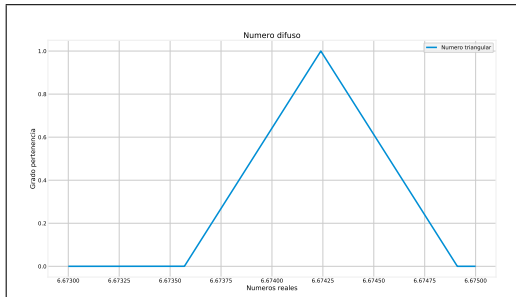
Y teniendo en cuenta la definición de seno, y coseno se puede escribir finalmente:

$$a_x = -\frac{GMx}{r^3} \quad \parallel \quad a_y = -\frac{GMy}{r^3}$$

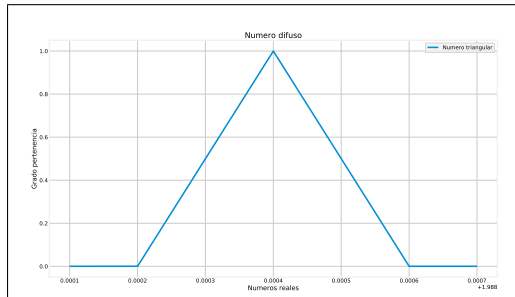
Y dado que por definición la aceleración es la segunda derivada de la ecuación de posición respecto al tiempo, se tiene:

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

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 (Se verá en el último capítulo la resolución numérica de este problema)

5.1.2. Aplicación a la mecánica cuántica

La ecuación de Schrödinger

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 [\[21\]](#)

John Glenn: "Pasar de una trayectoria elíptica a otra parabólica"

Katherine Johnson: "No se trata de una solución teórica, sino numérica" [22]

– 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 numérica de ecuaciones diferenciales hasta las técnicas más sofisticadas a nivel computacional.

En esta sección se va a hacer un análisis exhaustivo en las ventajas que nos ofrecen los distintos métodos con vistas a obtener el rendimiento más óptimo en cuanto a velocidad y en eficiencia energética.

Para entender esta sección sería conveniente revisar algunas de las técnicas mostradas en la sección anterior, y repasar las técnicas 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 soluciones:

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

6.1. EL MÉTODO DE EULER

Se considera ahora una discretación de x de manera que:

$$x_i = x_0 + ih$$

Donde h se llama paso del método de Euler, entonces se define la discretización de y como:

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

La definición de y_{i+1} se llama Método de Euler.

6.1.2. Método de Euler: Versión difusa

Se considera el siguiente problema difuso,

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

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:

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

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. Dado 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 se considera una partición dada por

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

6. RESOLUCIÓN NUMÉRICA DE ECUACIONES DIFERENCIALES DIFUSAS

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 la 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:

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

A continuación, se ofrecerá varias 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.

6.1. EL MÉTODO DE EULER

Implementación en Python

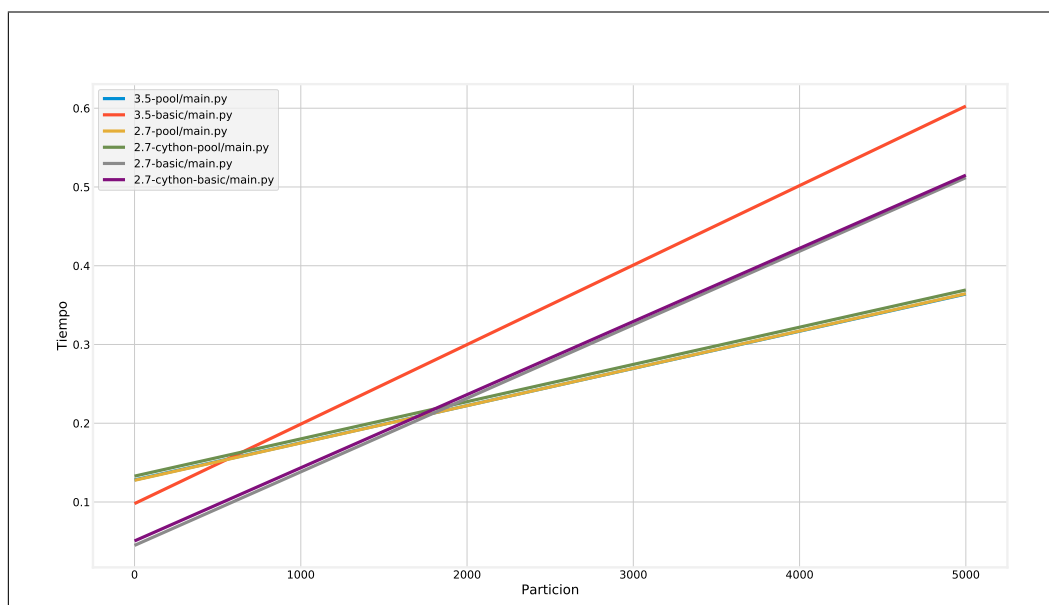


Figura 6.1: Distintos resultados en Python

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 más lento 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

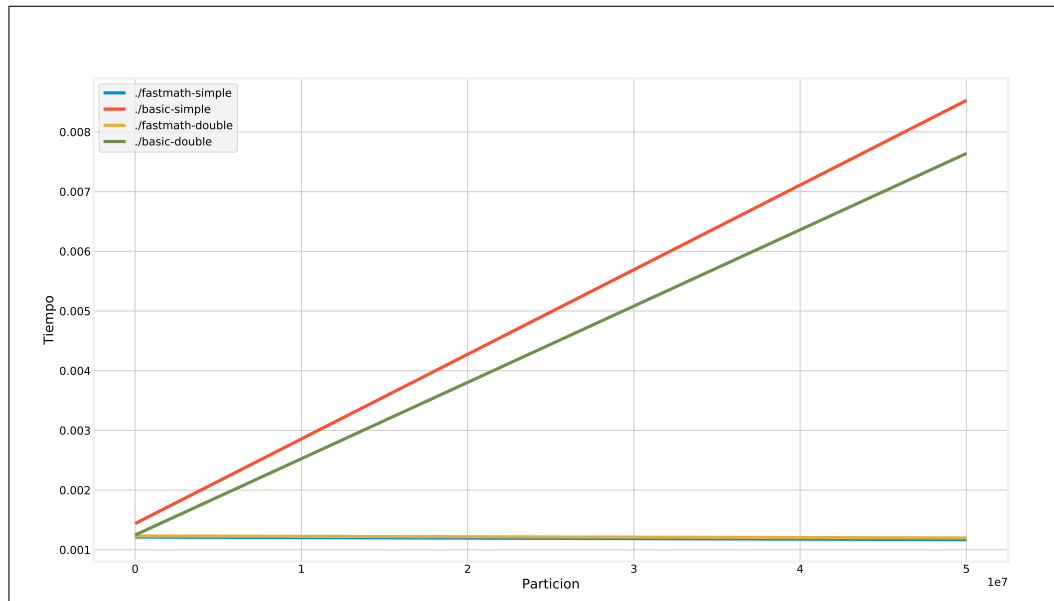


Figura 6.2: Distintos resultados en C

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 bastante 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	11.9 segundos
Fastmath doble	12.15 segundos
Doble	75.80 segundos
Simple	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.

6.1. EL MÉTODO DE EULER

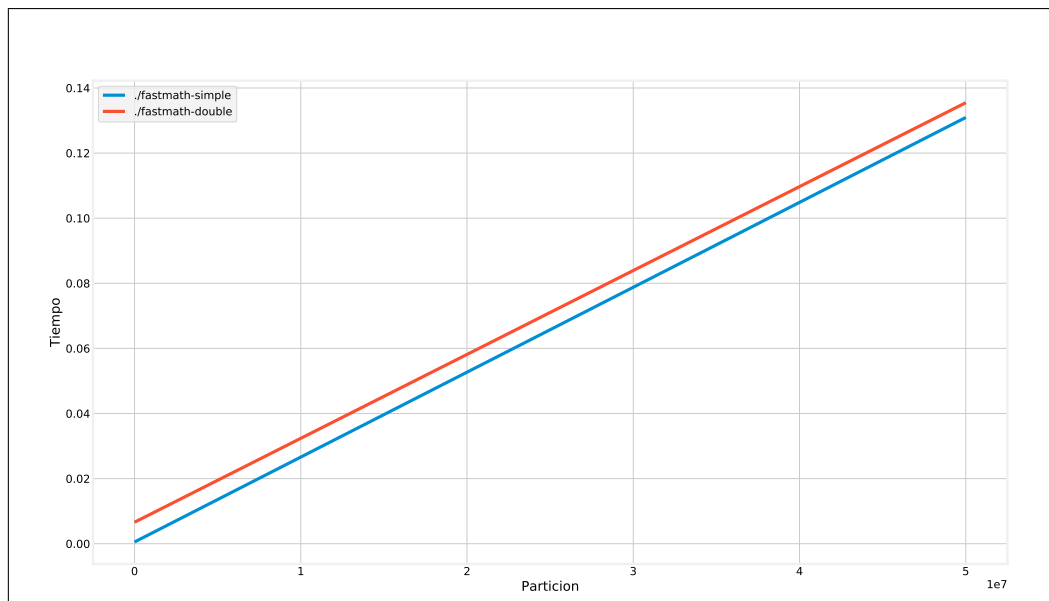


Figura 6.3: Distintos resultados en C

Aquí se pueden sacar unas 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	1278.03 segundos
Fastmath doble	1341.10 segundos

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

Test	Consumo energético
Fastmath simple	118,644028 $\mu A/s$
Fastmath doble	118,644058 $\mu A/s$
Simple	152,542389 $\mu A/s$
Doble	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, que como es bien 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:

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

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:

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

6.2. MÉTODO DE RUNGE-KUTTA DE CUARTO ORDEN

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

Se considera el siguiente problema difuso,

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

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:

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

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 tratar, se va a resolver el mismo problema anterior pero esta vez, aplicando 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.

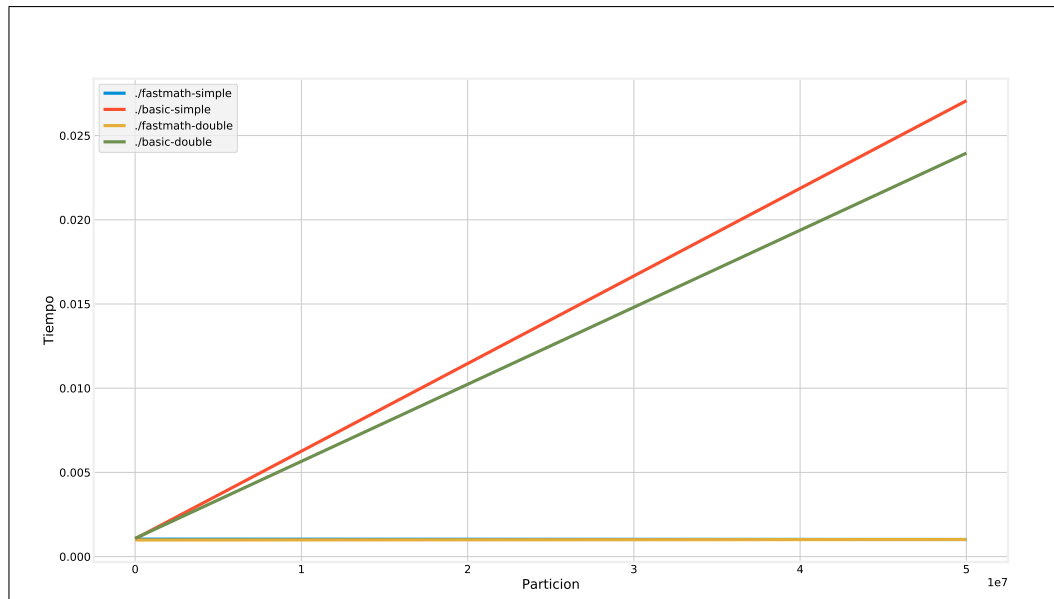


Figura 6.4: Distintos resultados en C

Test	Tiempo
Fastmath simple	10.1921305656 segundos
Fastmath doble	10.1441464424 segundos
Basic doble	238.943392515 segundos
Basic simple	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 información 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 comprueban las hipótesis del [Teorema de equivalencia entre EDO y EDD](#).
- 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](#)

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 una ecuación diferencial difusa sea equivalente a resolver una ecuación diferencial 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 aquí será tener la habilidad de poder resolver varios problemas.

Por tanto, las técnicas estudiadas aquí nacen de un 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 comprueban las hipótesis del [Teorema de equivalencia entre EDO y EDD](#).
- 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 hacía las ecuaciones en derivadas parciales difusa, 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 mejoras 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 [\[23\]](#)

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

```
#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

```
#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

```
#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

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

#include "math.h"

/**
 * For testing speed with different preccision.
 */
#ifndef 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);
    }
}
```

8. APÉNDICE: CÓDIGO FUENTE

```
    }

    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("Precision: %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_PRECISION
    gcc -o fastmath-double main.c -lm -Ofast -D DOUBLE_PRECISION
    gcc -o o3-double main.c -lm -O3 -D DOUBLE_PRECISION
```

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

```
#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("Precision: %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_PRECISION
    gcc -o fastmath-double main.c -lm -Ofast -D DOUBLE_PRECISION
    gcc -o o3-double main.c -lm -O3 -D DOUBLE_PRECISION
```

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 n) {
    return i * (n+1) + j;
}

edo_fuzzy_solver_error edo_fuzzy_solver_create(double * fuzzy_set, unsigned int fuzzy_set_length,
    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 tol,
    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_length, cudaMemcpyHostToDevice);
if (cuda_error != cudaSuccess) {
    error_t = EFS_E_CUDA_ERROR;
    goto clean;
}

// Allocate x values on GPU.
cuda_error = cudaMalloc(&x_values_cuda, sizeof(double) * solver.fuzzy_set_length);
if (cuda_error != cudaSuccess) {
    error_t = EFS_E_CUDA_ERROR;
    goto clean;
}

// Allocate y values on GPU.
cuda_error = cudaMalloc(&y_values_cuda, sizeof(double) * solver.fuzzy_set_length);
if (cuda_error != cudaSuccess) {
    error_t = EFS_E_CUDA_ERROR;
    goto clean;
}

// Calculate grid size.
cudaOccupancyMaxPotentialBlockSize(&minGridSize, &blockSize, edo_fuzzy_solver_kernel_runge_kutta);
gridSize = (solver.fuzzy_set_length + blockSize - 1) / blockSize;

// Run runge kutta in kernel.
edo_fuzzy_solver_kernel_runge_kutta << <gridSize, blockSize>> >(fuzzy_set_cuda, x_values_cuda, y_values_cuda, solver.fuzzy_set_length, solver.fuzzy_set_length);

// Synchronize device.
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);

// Copy x values from GPU to CPU.
cuda_error = cudaMemcpy(x_values_raw, x_values_cuda, solver.fuzzy_set_length * sizeof(double), cudaMemcpyDeviceToHost);
if (cuda_error != cudaSuccess) {
    error_t = EFS_E_CUDA_ERROR;
    goto clean;
}

// Copy y values from GPU to CPU.
cuda_error = cudaMemcpy(y_values_raw, y_values_cuda, solver.fuzzy_set_length * sizeof(double), cudaMemcpyDeviceToHost);
if (cuda_error != cudaSuccess) {
    error_t = EFS_E_CUDA_ERROR;
    goto clean;
}

// Reallocate solution arrays.
solution->x_values = (double**)realloc(solution->x_values, sizeof(double*) * n);
solution->y_values = (double**)realloc(solution->y_values, sizeof(double*) * n);
```

```

    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_length + 1) + j];
            solution->y_values[i][j] = y_values_raw[i * (solver.fuzzy_set_length + 1) + j];
        }
    }

    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_fuzzy_solver_t solver) {
    return NAN;
}

```


- [1] Barnabas Bede Luciana Takata Gomes, Laécio Carvalho de Barros. Fuzzy differential equations in various approaches. *Maths*, 1:11–38, 2015.
- [2] Scholarpedia. [Fuzzy sets](#).
- [3] Elena Medina Reus Jesus Medina Moreno. Apuntes de modelización matemática. *Maths*, 3:1–20, 2018.
- [4] D. Ralescu M. Puri. Fuzzy random variables. *Maths*, 1:409–422, 1986.
- [5] M. Hukuhara. Intégration des applications mesurables dont la valeur est un compact convexe. pages 205–223, 1967.
- [6] B. Bede. Mathematics of fuzzy sets and fuzzy logic (springer, berlin/heidelberg). 2013.
- [7] O. Kaleva. Fuzzy differential equations. fuzzy sets syst. 1987.
- [8] Birgit Ararat, Çağın; Rudloff. A characterization theorem for aumann integrals. 2014.
- [9] D. Ralescu M. Puri. Fuzzy random variables. 1986.
- [10] O. Kaleva. Fuzzy differential equations. 1987.
- [11] S.G. Gal B. Bede. Generalized differentiability of fuzzy-valued functions. 2013.
- [12] H. Román-Flores Y. Chalco-Cano. Comparison between some approaches to solve fuzzy differential equations. 2009.
- [13] H. Román-Flores Y. Chalco-Cano. Comparison between some approaches to solve fuzzy differential equations. 2009.
- [14] E. Hüllermeier. An approach to modeling and simulation of uncertain dynamical systems. 1997.

- [15] P. Diamond. Time-dependent differential inclusions, cocycle attractors and fuzzy differential equations.
- [16] P. Diamond. Brief note on the variation of constants formula for fuzzy differential equations.
- [17] José Carlos García Ortega. [Breve introducción a la programación en paralelo](#), 2018.
- [18] José Carlos García Ortega. [Experimentos numéricos](#), 2018.
- [19] dewar Linus Torvalds. What is acceptable for -ffast-math? (was: associative law incombine). 2001.
- [20] NASA. [ASTRONOMICAL CONSTANTS](#).
- [21] 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.
- [22] Katherine G. Skopinski, T. H. & Johnson. Determination of azimuth angle at burnout for placing a satellite over a selected earth position. 1960.
- [23] Shadan Sadigh Behzadi. Solving cauchy reaction-diffusion equation by using picard method. 2013.
- [24] Samuel Corveleyn. The numerical solution of elliptic partial differential equations with fuzzy coefficients. 2014.
- [25] Douglas C. Giancoli. *Physics for Scientists and Engineers*. Pearson, 2014.