Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Matemáticas e Informática

Trabajo Fin de Grado

Desarrollo de una aplicación didáctica en Cálculo: resolución numérica de integrales

Autor: David García Sanz
Tutor: Javier López de la Cruz
Cotutor: Juan Ángel Rojo Carulli

Madrid, Mayo - 2023

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Matemáticas e Informática

Título: Desarrollo de una aplicación didáctica en Cálculo: resolución numérica de integrales

Mayo - 2023

Autor: David García Sanz
Tutor: Javier López de la Cruz
Departamento de Matemática Aplicada a las TIC
ETSI Informáticos
Universidad Politécnica de Madrid
Cotutor: Juan Ángel Rojo Carulli
Departamento de Matemática Aplicada a las TIC
ETSI Informáticos
Universidad Politécnica de Madrid

Tabla de contenidos

Resumen

Este Trabajo Fin de Grado se engloba dentro del denominado “Proyecto Calculadora” que se desarrolla como parte de la actividad del Grupo de Innovación Educativa “Desarrollo de Tecnologías en la Enseñanza de las Matemáticas” en la Universidad Politécnica de Madrid.

Dicho proyecto se centra en la creación de una aplicación didáctica que sirva como herramienta de apoyo al aprendizaje de las asignaturas de Matemáticas para los estudiantes del Grado en Matemáticas e Informática, que se imparte actualmente en la Escuela Técnica Superior de Ingeniería Informática de la Universidad Politécnica de Madrid. Para ello, se está llevando a cabo un proyecto en el cual se pretende implementar, mediante algoritmos, la resolución de ejercicios de las asignaturas de Matemáticas del grado mencionado anteriormente.

Más concretamente, el presente trabajo se enfoca en la implementación de algoritmos de resolución numérica de integrales, con el fin de que los estudiantes puedan practicar con ayuda de un software especializado, y así, mejorar tanto su aprendizaje como el desarrollo de las competencias de las diferentes asignaturas en las que se aborden los conocimientos que aquí se desarrollan.

Abstract

This Final Degree Project is part of the so-called “Proyecto Calculadora” developed as part of the activities of the Educational Innovation Group “Desarrollo de Tecnologías en la Enseñanza de las Matemáticas” at the Universidad Politécnica de Madrid.

The project focuses on creating an educational application that serves as a learning support tool for Mathematics subjects for students in the Mathematics and Computer Science degree program, currently taught at the Universidad Politécnica de Madrid. To achieve this, a project is being carried out to implement, through algorithms, the solution of exercises from the aforementioned Mathematics subjects.

Specifically, this work focuses on the implementation of numerical integration algorithms, so that students can practice using specialized software, thereby improving both their learning and the development of competencies in the different subjects that address the knowledge presented here.

Capítulo 1

Introducción

Este Trabajo Fin de Grado se engloba dentro del denominado “Proyecto Calculadora” que se desarrolla como parte de la actividad del Grupo de Innovación Educativa “Desarrollo de Tecnologías en la Enseñanza de las Matemáticas” en la Universidad Politécnica de Madrid.

Dicho proyecto se centra en la creación de una aplicación didáctica que sirva como herramienta de apoyo al aprendizaje de las asignaturas de Matemáticas para los estudiantes del Grado en Matemáticas e Informática, que se imparte actualmente en la Escuela Técnica Superior de Ingeniería Informática de la Universidad Politécnica de Madrid. Para ello, se está llevando a cabo un proyecto en el cual se pretende implementar, mediante algoritmos, la resolución de ejercicios de las asignaturas de Matemáticas del grado mencionado anteriormente.

Más concretamente, el presente trabajo se enfoca en la implementación de algoritmos de resolución numérica de integrales, con el fin de que los estudiantes puedan practicar con ayuda de un software especializado, y así, mejorar tanto su aprendizaje como el desarrollo de las competencias de las diferentes asignaturas en las que se aborden los conocimientos que aquí se desarrollan.

De esta forma, proporcionaremos los códigos necesarios para llevar a cabo diferentes algoritmos sobre integración numérica, códigos que deben ser implementados en una aplicación web que está siendo a su vez diseñada y creada por otros estudiantes, bajo la supervisión de los miembros del Grupo de Innovación Educativa al que hemos hecho referencia previamente.

En concreto, en este trabajo se abordarán los siguientes métodos de integración numérica:

- Fórmula de los rectángulos compuesta.
- Fórmula del punto medio compuesta.
- Fórmula del trapecio compuesta.
- Fórmula de Simpson compuesta.
- Fórmula del trapecio recursiva.
- Método de Romberg.
- Método adaptativo de Simpson.

La implementación de dichos algoritmos se ha llevado a cabo en el lenguaje de programación Python, con el objetivo de hacerlos más accesibles y fáciles de utilizar para cualquier usuario. Asimismo, en esta memoria se explicará el funcionamiento de cada uno de los algoritmos, así como las bases teóricas en las que se sustentan.

Por último, también se ha llevado a cabo un proceso de armonización del código con la aplicación web desarrollada, de modo que el resultado sea operativo y funcione correctamente.

Capítulo 2

Marco Teórico

La integración numérica es una herramienta que proporcionan las Matemáticas y que nos permite obtener fórmulas que nos ayudan a calcular de forma aproximada integrales definidas. Esto es especialmente útil en aquellos casos en los que la integral definida no puede obtenerse analíticamente y, además, nos permite hacer uso de ordenadores para realizar todos los cálculos de forma bastante precisa y en relativamente poco tiempo.

El contenido de este capítulo está basado, esencialmente, en las referencias [1, 2, 3], donde pueden encontrarse además demostraciones rigurosas de los resultados que se comentan en las siguientes secciones y ejemplos detallados sobre cada uno de los métodos de integración numérica que se abordan.

2.1. Fórmulas de cuadratura. Orden

Las fórmulas para estimar el valor de una integral definida se denominan fórmulas de cuadratura. En su forma más simple, estas fórmulas aproximan el área bajo una curva por el área de un paralelogramo. Sin embargo, esta aproximación solo es precisa cuando la base del paralelogramo es pequeña. Por lo tanto, las fórmulas más efectivas estiman la integral mediante la suma de áreas de varios paralelogramos con bases pequeñas.

En general, las fórmulas de cuadratura se pueden escribir en la forma:

$$I^*(f) = \sum_{k=1}^{n} \alpha_k f(x_k) \tag{2.1}$$

variando unas de otras en la forma de elegir los puntos $x_1 < \ldots < x_n$ en el intervalo $[a, b]$ y los coeficientes (también llamados pesos) α_k, donde $k = 1, \ldots, n$.

Para determinar el grado de exactitud de una fórmula de cuadratura, es decir, el error que se comete al sustituir la integral definida por la suma finita,

$$E(f) = I(f) - I^*(f) = \int_a^b f(x)\mathrm{d}x - \sum_{k=1}^{n} \alpha_k f(x_k) \tag{2.2}$$

se suele utilizar el concepto de orden.

Se dice que una fórmula de cuadratura es de orden $m \in \mathbb{N}$ (o que es exacta para polinomios de grado $m \in \mathbb{N}$) si el error de dicha fórmula verifica que $E(x^k) = 0$ para todo $k = 0, \ldots, m$ y $E(x^{m+1}) \neq 0$.

Esto implica que la fórmula en particular produce el resultado preciso de la integral definida si se aplica a una función f que es un polinomio de grado igual o inferior a m, pero no necesariamente proporciona el resultado exacto para polinomios de grado superior a m.

2.2. Fórmulas de cuadratura elementales

En este apartado se expondrán las cuatro fórmulas de cuadratura elementales en las que se sustenta este trabajo:

1. **Formula de los Rectángulos**

   La fórmula de cuadratura mas sencilla es la fórmula de los rectángulos:

   $$\int_a^b f(x)\mathrm{d}x \approx I_1(f) = (b-a)f(a), \qquad \int_a^b f(x)\mathrm{d}x \approx I_2(f) = (b-a)f(b). \tag{2.3}$$

   En el primer caso se aproxima la integral por el area del rectángulo de base $[a, b]$ y altura $f(a)$ y en el segundo por el de altura $f(b)$. Ambas son de orden cero, es decir, exactas para polinomios constantes.

2. **Fórmula del Punto Medio**

   La fórmula elemental del punto medio también usa el área de un rectángulo pero tomando como altura el valor de f en el punto medio del intervalo:

$$\int_a^b f(x)\mathrm{d}x \approx I_3(f) = (b-a)f\left(\frac{a+b}{2}\right). \tag{2.4}$$

Esta fórmula es de orden 1, a modo de ejemplo:

$$E(1) = \int_a^b \mathrm{d}x - I_3(1) = (b-a) - (b-a) = 0, \tag{2.5}$$

$$E(x) = \int_a^b x\mathrm{d}x - I_3(x) = \frac{1}{2}(b^2-a^2) - \frac{(b-a)(a+b)}{2} = 0, \tag{2.6}$$

$$E(x^2) = \int_a^b x^2\mathrm{d}x - I_3(x^2) = \frac{1}{3}(b^3-a^3) - \frac{(b^2-a^2)(a+b)}{4} \neq 0. \tag{2.7}$$

Por tanto queda comprobado que la fórmula del punto medio es de orden 1.

3. **Fórmula del Trapecio**

   La siguiente fórmula elemental es la del trapecio, la integral se aproxima por el área de un trapecio, esta fórmula también es de orden 1.

$$\int_a^b f(x)\mathrm{d}x \approx I_4(f) = \frac{(b-a)}{2}(f(a)+f(b)), \tag{2.8}$$

$$E(1) = \int_a^b \mathrm{d}x - I_4(1) = (b-a) - \left[\frac{(b-a)}{2}2\right] = 0, \tag{2.9}$$

$$E(x) = \int_a^b x\mathrm{d}x - I_4(x) = \left[\frac{1}{2}(b^2-a^2)\right] - \left[\frac{(b-a)}{2}(a+b)\right] = 0, \tag{2.10}$$

$$E(x^2) = \int_a^b x^2\mathrm{d}x - I_4(x^2) = \left[\frac{1}{3}(b^3-a^3)\right] - \left[(b^2-a^2)\frac{(b-a)}{2}\right] \neq 0. \tag{2.11}$$

Por tanto queda comprobado que la formula del trapecio es de orden 1.

4. **Fórmula de Simpson**

   La última de las fórmulas elementales es la fórmula de Simpson donde se aproxima la integral de f por el área encerrada bajo un arco de parábola que coincide con f en tres puntos: los extremos del intervalo $[a, b]$ y su punto medio (véase Figura 2.1):

$$\int_a^b f(x)\mathrm{d}x \approx I_5(f) = \frac{(b-a)}{6}\left((f(a)+4f(\frac{a+b}{2})+f(b))\right). \tag{2.12}$$

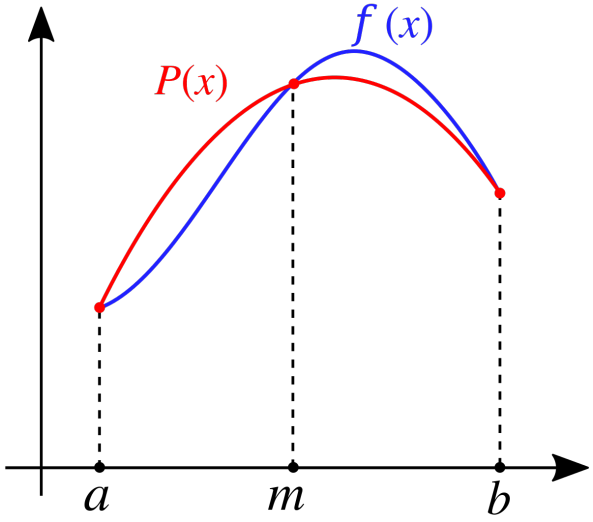

Figura 2.1: Ejemplo de arco de parábola (en rojo) para usar la fórmula de Simpson y aproximar el área bajo la función $f(x)$ en $[a, b]$.

Razonando es posible deducir que la fórmula elemental de Simpson de orden 3.

Podríamos continuar obteniendo nuevas fórmulas de cuadratura similares a las presentadas en esta sección. Para ello basta usar el valor de la función cuya área se quiere aproximar en más puntos o bien elegir dichos puntos de forma que la fórmula de cuadratura obtenida tenga el mayor orden posible. Para más información, véase [1] y [2].

2.3. Fórmulas de cuadratura compuestas

Si el número de puntos elegidos en el intervalo $[a, b]$ se incrementa, las fórmulas de cuadratura simples mencionadas anteriormente no suelen brindar estimaciones de la integral muy precisas. Por consiguiente, en la práctica se utilizan las fórmulas de cuadratura compuestas, las cuales descomponen la integral definida en una suma de integrales sobre subintervalos "pequeños" y aplican las fórmulas previas en cada uno de ellos.

Sean $a = x_1 < x_2 < \ldots < x_n = b$ un conjunto de puntos en el intervalo $[a, b]$. Por las propiedades de la integral se tiene:

$$\int_a^b f(x)\mathrm{d}x = \int_{x_1}^{x_2} f(x)\mathrm{d}x + \int_{x_2}^{x_3} f(x)\mathrm{d}x + \ldots + \int_{x_{n-1}}^{x_n} f(x)\mathrm{d}x = \sum_{i=1}^{n-1} \int_{x_i}^{x_{i+1}} f(x)\mathrm{d}x. \tag{2.13}$$

Usando alguna de las fórmulas elementales en cada subintervalo:

$$\int_a^b f(x)\mathrm{d}x = \sum_{i=1}^{n-1} \int_{x_i}^{x_{i+1}} f(x)\mathrm{d}x \approx \sum_{i=1}^{n-1} I_k(f; [x_i, x_{i+1}]), \tag{2.14}$$

donde $I_k(f; [x_i, x_{i+1}])$ denota una cierta fórmula de cuadratura elemental de orden k aplicada para aproximar la integral de f en el intervalo $[x_i, x_{i+1}]$.

En las siguientes secciones se presentarán individualmente cada una de las fórmulas de cuadratura compuesta.

2.3.1. Formula de los Rectángulos Compuesta

La primera fórmula de integración numérica compuesta que se elaborará en este Trabajo Fin de Grado es la Fórmula de los Rectángulos Compuesta, que se apoya en la fórmula de cuadratura elemental de los rectángulos. En ella podemos distinguir dos casos, con subintervalos con igual longitud o subintervalos con distinta longitud:

- Con subintervalos de distinta longitud:

$$\int_a^b f(x)\mathrm{d}x \approx I_1^c(f) = \sum_{i=1}^{n-1} I_k(f; [x_i, x_{i+1}]) = \sum_{i=1}^{n-1} (x_{i+1} - x_i) f(x_i), \tag{2.15}$$

$$\int_a^b f(x)\mathrm{d}x \approx I_2^c(f) = \sum_{i=1}^{n-1} I_2(f; [x_i, x_{i+1}]) = \sum_{i=1}^{n-1} (x_{i+1} - x_i) f(x_{i+1}). \tag{2.16}$$

- Con subintervalos de misma longitud (véanse Figura 2.2 y Figura 2.3):

$$I_1^c(f) = \sum_{i=1}^{n-1} (x_{i+1} - x_i) f(x_i) = \sum_{i=1}^{n-1} h f(x_i) = h \sum_{i=1}^{n-1} f(x_i), \tag{2.17}$$

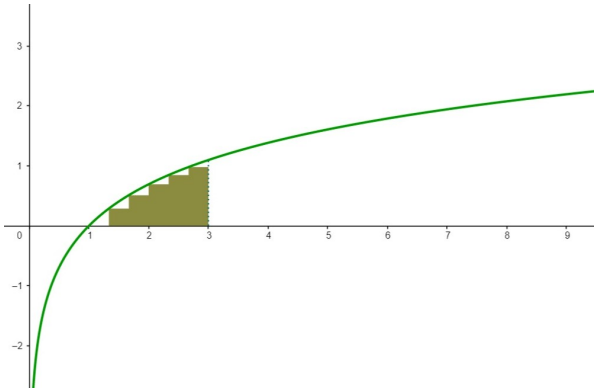

Figura 2.2: Fórmula de los rectángulos compuesta I_1^c con subintervalos de igual longitud para aproximar la integral de $f(x) = \ln(x)$ en el intervalo $[1, 3]$.

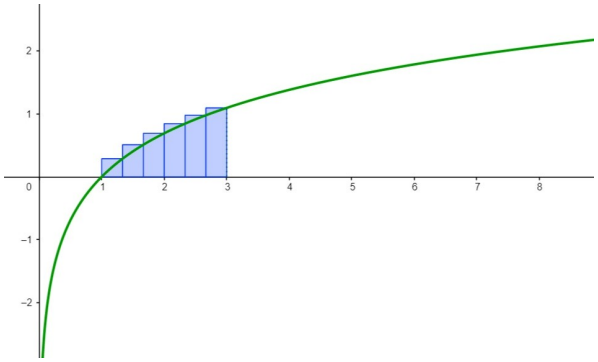

Figura 2.3: Fórmula de los rectángulos compuesta I_2^c con subintervalos de igual longitud para aproximar la integral de $f(x) = \ln(x)$ en el intervalo $[1, 3]$.

$$I_2^c(f) = \sum_{i=1}^{n-1}(x_{i+1} - x_i)f(x_{i+1}) = \sum_{i=1}^{n-1} hf(x_{i+1}) = h\sum_{i=1}^{n-1} f(x_{i+1}). \tag{2.18}$$

En ambos casos se divide el intervalo en subintervalos y se aplica la fórmula de rectángulos a cada uno de ellos, sumando los resultados.

2.3.2. Fórmula del Punto Medio Compuesta

La segunda fórmula de integración numérica compuesta que se elaborará en este Trabajo Fin de Grado es la Fórmula del Punto Medio Compuesta, que se apoya en la fórmula de cuadratura elemental del punto medio. Respecto a su fórmula compuesta:

$$\int_a^b f(x)\mathrm{d}x \approx I_3^c(f) = \sum_{i=1}^{n-1} I_3(f;[x_i,x_{i+1}]) = \sum_{i=1}^{n-1}(x_{i+1}-x_i)f\left(\frac{x_i+x_{i+1}}{2}\right). \quad (2.19)$$

En caso de que los intervalos tengan igual longitud la fórmula del punto medio compuesto sería:

$$\int_a^b f(x)\mathrm{d}x \approx I_3^c(f) = \sum_{i=1}^{n-1} hf\left(\frac{x_i+x_{i+1}}{2}\right) = h\sum_{i=1}^{n-1} f\left(\frac{x_i+x_{i+1}}{2}\right). \quad (2.20)$$

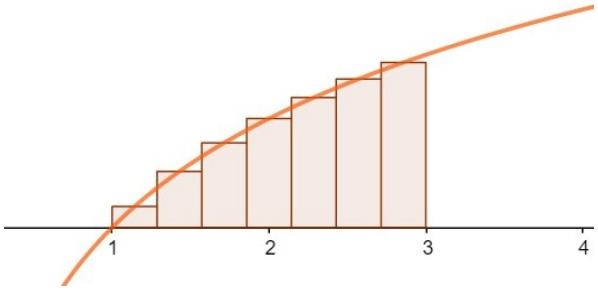

Figura 2.4: Fórmula del punto medio compuesta para aproximar la integral de $f(x) = \ln(x)$ en el intervalo $[1, 3]$.

La fórmula del punto medio compuesta divide el intervalo $[a, b]$ en subintervalos de menor longitud y aplica la fórmula del punto medio elemental en cada uno de dichos subintervalos (véase Figura 2.4).

2.3.3. Fórmula de los Trapecios Compuesta

La tercera fórmula de integración numérica compuesta que se elaborará en este Trabajo Fin de Grado es la Fórmula de los Trapecios Compuesta, que se apoya en la fórmula de cuadratura elemental del trapecio. Respecto a su fórmula compuesta queda así:

$$\int_a^b f(x)\mathrm{d}x \approx I_4^c(f) = \sum_{i=1}^{n-1} I_4(f;[x_i,x_{i+1}]) = \sum_{i=1}^{n-1}(x_{i+1}-x_i)\frac{f(x_i)+f(x_{i+1})}{2}. \quad (2.21)$$

En caso de que los subintervalos sean de igual longitud:

$$I_4^c(f) = \sum_{i=1}^{n-1}(x_{i+1} - x_i)\frac{f(x_i) + f(x_{i+1})}{2} = \sum_{i=1}^{n-1} h\frac{f(x_i) + f(x_{i+1})}{2}$$
$$= \frac{h}{2}\sum_{i=1}^{n-1} h(f(x_i) + f(x_{i+1})) = \frac{h}{2}\left(f(x_1) + 2\sum_{i=2}^{n-1} f(x_i) + f(x_n)\right). \tag{2.22}$$

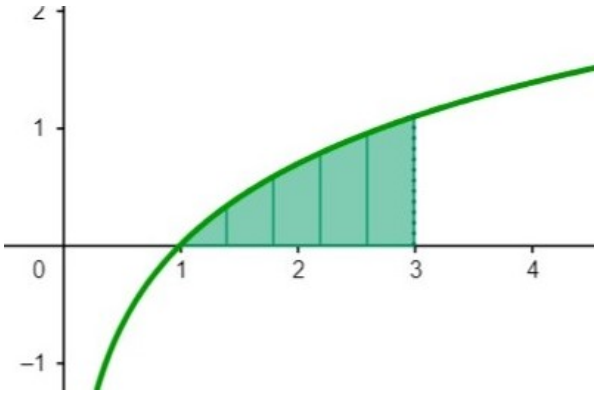

Figura 2.5: Fórmula del trapecio compuesta para aproximar la integral de $f(x) = \ln(x)$ en el intervalo $[1, 3]$.

La fórmula del trapecio compuesta consiste en aproximar el área bajo la curva de una función aplicando la correspondiente fórmula elemental a subintervalos del intervalo original (véase Figura 2.5).

Error de Análisis

Sea el intervalo $[a, b]$ dividido en M subintervalos $[x_k, x_{k+1}]$ de longitud $h = (b - a)/M$. La fórmula del trapecio compuesta puede escribirse como:

$$T(f, h) = \frac{h}{2}(f(a) + f(b) + h\sum_{k=1}^{M-1} f(x_k), \tag{2.23}$$

que es una aproximación de la integral:

$$\int_a^b f(x)\mathrm{d}x = T(f, h) + E_T(f, h), \tag{2.24}$$

donde $E_T(f, h)$ denota el error al aproximar la integral de f en $[a, b]$ usando la fórmula de los trapecios compuesta con todos los subintervalos de longitud h.

Además, puede demostrarse que si $f \in C^2[a,b]$ existe un valor c con $a < c < b$ de tal forma que el error $E_T(f,h)$ viene dado por:

$$E_T(f,h) = \frac{-(b-a)f^{(2)}(c)h^2}{12} = O(h^2). \tag{2.25}$$

2.3.4. Fórmula de Simpson Compuesta

La cuarta fórmula de integración numérica compuesta que se elaborará en este Trabajo Fin de Grado es la Fórmula de Simpson Compuesta, que se apoya en la fórmula de cuadratura elemental de Simpson. Respecto a su fórmula compuesta se escribe:

$$\begin{aligned}\int_a^b f(x)\mathrm{d}x \approx I_5^c(f) &= \sum_{i=1}^{n-1} I_5(f;[x_i,x_{i+1}]) \\ &= \sum_{i=1}^{n-1} \frac{(x_{i+1}-x_i)}{6}\left(f(x_i) + 4f\left(\frac{x_i+x_{i+1}}{2}\right) + f(x_{i+1})\right).\end{aligned} \tag{2.26}$$

En el caso de subintervalos de igual longitud, se transforma en:

$$\begin{aligned}I_5^c(f) &= \sum_{i=1}^{n-1} \frac{h}{6}\left(f(x_i) + 4f\left(\frac{x_i+x_{i+1}}{2}\right) + f(x_{i+1})\right) \\ &= \frac{h}{6}\sum_{i=1}^{n-1}\left(f(x_i) + 4f\left(\frac{x_i+x_{i+1}}{2}\right) + f(x_{i+1})\right) \\ &= \frac{h}{6}\left[f(x_1) + 2\sum_{i=2}^{n-1} f(x_i) + f(x_n) + 4\sum_{i=1}^{n-1} f\left(\frac{x_i+x_{i+1}}{2}\right)\right],\end{aligned} \tag{2.27}$$

donde h es la longitud de cada subintervalo, es decir, $h = (b-a)/n$, x_i son los puntos de la partición del intervalo $[a,b]$, es decir, $x_i = a + ih$, y n es el número total de subintervalos.

2.3.4.1. Fórmula Alternativa

Dado el intervalo $[a,b]$ dividido en $2M$ subintervalos $[x_k, x_{k+1}]$ de longitud $h = (b-a)/2M$. La fórmula de Simpson compuesta para $2M$ sub-intervalos se puede expresar como:

$$S(f,h) = \frac{h}{3}\sum_{k=1}^{M}(f(x_{2k-2}) + 4f(x_{2k-1}) + f(x_{2k}))$$
$$= \frac{h}{3}(f(a)+f(b)) + \frac{2h}{3}\sum_{k=1}^{M-1} f(x_{2k}) + \frac{4h}{3}\sum_{k=1}^{M} f(x_{2k-1}). \quad (2.28)$$

En la regla de Simpson compuesta, se divide el intervalo $[a,b]$ en n subintervalos y se aplica la regla de Simpson elemental a cada subintervalo. La aproximación de la integral en el intervalo $[a,b]$ se obtiene sumando las aproximaciones en cada subintervalo.

La fórmula de Simpson compuesta es más precisa que la regla del trapecio compuesta para el mismo número de puntos, ya que utiliza una aproximación polinómica de segundo grado, en lugar de una aproximación lineal.

Análisis del error

Sea el intervalo $[a,b]$ dividido en $2M$ sub-intervalos $[x_k, x_{k+1}]$ de tamaño $h = (b-a)/2M$. La fórmula de Simpson compuesta es:

$$S(f,h) = \frac{h}{3}(f(a)+f(b)) + \frac{2h}{3}\sum_{k=1}^{M-1} f(x_{2k}) + \frac{4h}{3}\sum_{k=1}^{M} f(x_{2k-1}) \quad (2.29)$$

que aproxima la integral:

$$\int_a^b f(x)\mathrm{d}x = S(f,h) + E_S(f,h), \quad (2.30)$$

donde $E_S(f,h)$ denota el error que se comete al aproximar la integral de f en el intervalo $[a,b]$ haciendo uso de la fórmula de Simpson compuesta con subintervalos de igual longitud h.

Además, si $f \in C^4[a,b]$ entonces existe un valor c con $a < c < b$ tal que el error $E_T(f,h)$ viene dado como:

$$E_T(f,h) = \frac{-(b-a)f^{(4)}(c)h^4}{12} = O(h^4). \quad (2.31)$$

2.3.5. Fórmula de los Trapecios Recursiva

A continuación, una vez que hemos conocido las fórmulas de cuadratura compuestas en base a las fórmulas de cuadratura elementales, se expondrán nuevas fórmulas y métodos más precisos que las ya mencionadas.

En esta sección se describe cómo calcular aproximaciones de Simpson mediante una combinación especial de reglas trapezoidales, con el objetivo de obtener una mayor precisión al utilizar un mayor número de subintervalos. Para determinar cuántos subintervalos se deben utilizar, se emplea un proceso secuencial que comienza con dos subintervalos y se va incrementando hasta alcanzar la precisión deseada.

Para ello, se genera una secuencia de aproximaciones de las reglas trapezoidales. Cada vez que se duplica el número de subintervalos, se duplica también aproximadamente el número de valores de la función, ya que es necesario evaluar la función en todos los puntos anteriores y en los puntos medios de los subintervalos anteriores. Este método resulta útil para optimizar el cálculo de las aproximaciones de Simpson y obtener resultados precisos.

Dado un $J \geq 1$ y dados los puntos $x_k = a + kh$ que subdividen el intervalo $[a, b]$ en $2^J = 2M$ subintervalos de misma longitud $h = (b-a)/2^J$. Las fórmulas del trapecio $T(f, h)$ y $T(f, 2h)$ cumplen la relación:

$$T(f,h) = \frac{T(f,2h)}{2} + h\sum_{k=1}^{M} f(x_{2k-1}). \tag{2.32}$$

Ahora pasamos a dividir el intervalo $[a, b]$ en el número de subintervalos deseados en función de la precisión deseada. Para ello definimos $T(0) = (h/2)(f(a)+f(b))$ que es la fórmula del trapecio en el intervalo $[a, b]$ de longitud $h = b - a$. Después para cada $J \geq 1$ definimos $T(J) = T(f, h)$, donde $T(f, h)$ es la regla del trapecio sobre un intervalo de longitud $h = (b - a)/2^J$.

Por tanto, comenzando por $T(0)$, la sucesión de fórmulas del trapecio $T(J)$ se calcula con la siguiente fórmula recursiva:

$$T(J) = \frac{T(J-1)}{2} + h\sum_{k=1}^{M}(f(x_{2k-1}) \quad \text{para } J = 1, 2, \ldots \tag{2.33}$$

donde $h = (b - a)/2^J$ y $x_k = a + kh$.

2.3.5.1. Relación con la fórmula de Simpson

Cuando se usa la fórmula del trapecio utilizando longitudes de intervalo $2h$ y h, el resultado es $T(f, 2h)$ y $T(f, h)$, respectivamente. Estos valores se

combinan para obtener la regla de Simpson:

$$S(f,h) = \frac{4T(f,h) - T(f,2h)}{3}. \tag{2.34}$$

Sea ahora $\{T(J)\}$ la sucesión de fórmulas del trapecio generadas por recursividad. Si $J \geq 1$ y $S(J)$ es la fórmula de Simpson para 2^J subintervalos de $[a,b]$, entonces se tiene la siguiente relación:

$$S(J) = \frac{4T(J) - T(J-1)}{3} \quad \text{para } J = 1, 2, \ldots \tag{2.35}$$

2.3.6. Método de Romberg

En las secciones anteriores vimos que los términos de error $E_T(f,h)$ $E_S(f,h)$ para la fórmula del trapecio compuesta y la fórmula de Simpson compuesta son de orden $O(h^2)$ y $O(h^4)$, respectivamente. Por lo tanto:

$$\int_a^b f(x)\mathrm{d}x = T(f,h) + O(h^2), \tag{2.36}$$

$$\int_a^b f(x)\mathrm{d}x = S(f,h) + O(h^4). \tag{2.37}$$

Supongamos que se utiliza una fórmula de cuadratura con longitudes de intervalo h y $2h$ y luego se lleva a cabo una manipulación algebraica de las dos para producir una aproximación mejorada. Cada nivel sucesivo de mejora aumenta el orden del término de error de $O(h^{2N})$ a $O(h^{2N+2})$. Este proceso es llamado integración de Romberg.

La ventaja del método de Romberg es que todos los pesos son positivos y las abscisas igualmente espaciadas son fáciles de calcular. Sin embargo, una desventaja computacional de la integración de Romberg es que se requiere el doble de evaluaciones de las funciones para reducir el error de $O(h^{2N})$ a $O(h^{2N+2})$. Para mantener el número de cálculos bajo, se pueden utilizar reglas secuenciales. El desarrollo de la integración de Romberg se basa en la suposición teórica de que, si $f \in C^N[a,b]$ para todo N, entonces el término de error para la fórmula del trapecio compuesta se puede representar en una serie que solo involucra potencias pares de h:

$$\int_a^b f(x)\mathrm{d}x = T(f,h) + E_T(f,h), \tag{2.38}$$

donde

$$E_T(f,h) = a_1h^2 + a_2h^4 + a_3h^6 + \ldots \tag{2.39}$$

2.3.6.1. Integración con mejora de Richardson

La mejora de Richardson para la integración de Romberg es un proceso iterativo que implica el cálculo de fórmulas de cuadratura mejoradas mediante la eliminación de términos de error de órdenes pares sucesivos. El proceso comienza con la aplicación de la regla trapezoidal con dos longitudes de intervalo diferentes, y luego se utiliza la manipulación algebraica de las dos aproximaciones para obtener otra mejorada. Este proceso se repite sucesivamente para eliminar términos de error de órdenes pares cada vez más altos. Como resultado, se generan fórmulas de cuadratura que solo involucran potencias pares de h, lo que permite obtener una convergencia más rápida y precisa. Este proceso se conoce como integración de Romberg con mejora de Richardson. La primera mejora que se obtiene con este proceso es la regla de Simpson para $2M$ intervalos, utilizando como punto de partida las fórmulas $T(f, 2h)$ y $T(f, h)$ y las ecuaciones:

$$\int_a^b f(x)\mathrm{d}x = T(f, 2h) + a_1 4h^2 + a_2 16h^4 + a_3 64h^6 + \dots \tag{2.40}$$

y

$$\int_a^b f(x)\mathrm{d}x = T(f, h) + a_1 h^2 + a_2 h^4 + a_3 h^6 + \dots \tag{2.41}$$

Multiplicando (2.41) por 4 obtenemos:

$$4\int_a^b f(x)\mathrm{d}x = 4T(f, h) + a_1 4h^2 + a_2 4h^4 + a_3 4h^6 + \dots \tag{2.42}$$

Eliminando a_1 restando (2.41) a (2.42) obtenemos:

$$3\int_a^b f(x)\mathrm{d}x = 4T(f, h) - T(f, 2h) - a_2 12h^4 - a_3 60h^6 - \dots \tag{2.43}$$

Dividimos (2.43) entre 3 y renombramos los coeficientes:

$$\int_a^b f(x)\mathrm{d}x = \frac{4T(f, h) - T(f, 2h)}{3} + b_1 h^4 + b_2 h^6 + \dots \tag{2.44}$$

Aplicando la relación que se muestra en el apartado 2.3.5.1, obtenemos:

$$\int_a^b f(x)\mathrm{d}x = S(f, h) + b_1 h^4 + b_2 h^6 + \dots \tag{2.45}$$

Por tanto, queda demostrado que $E_S(f, h)$ involucra solo potencias pares de h.

El patrón general para la integración de Romberg es como sigue. Consideremos dos aproximaciones $R(2h, K-1)$ Y $R(h, K-1)$ de la cantidad Q de tal forma que se tiene

$$Q = R(h, K-1) + c_1 h^{2K} + c_2 h^{2K+2} + \dots \tag{2.46}$$

y

$$Q = R(2h, K-1) + c_1 4^K h^{2K} + c_2 4^{K+1} h^{2K+2} + \dots \tag{2.47}$$

Una aproximación mejorada será de la siguiente forma:

$$Q = \frac{4^K R(h, K-1) - R(2h, K-1)}{4^K - 1} + O(h^{2K+2}). \tag{2.48}$$

Definiendo la sucesión de fórmulas de cuadratura para $f(x)$ en el intervalo $[a, b]$ como:

$$R(J, 0) = T(J) \quad \text{para } J \geq 0, \tag{2.49}$$

$$R(J, 1) = S(J) \quad \text{para } J \geq 1, \tag{2.50}$$

donde $\{T(J)\}$ y $\{S(J)\}$ son las correspondientes sucesiones de fórmulas de los trapecios y de Simpson, respectivamente.

La sucesión $\{R(J, 0)\}$ se usa para generar la primera mejora $\{R(J, 1)\}$. La fórmula general para construir mejoras viene dada entonces como:

$$R(J, K) = \frac{4^K R(h, K-1) - R(J-1, K-1)}{4^K - 1} \quad \text{para } J \geq K. \tag{2.51}$$

2.3.6.2. Precision de la integración de Romberg

Dada $f \in C^{2K+2}[a, b]$, el error del método de Romberg viene dado por la fórmula:

$$\int_a^b f(x)\mathrm{d}x = R(J, K) + b_K h^{2K+2} f^{2K+2)}(c_{J,K}) = R(J, K) + O(h^{2K+2}), \tag{2.52}$$

donde $h = (b-a)/2^J$, b_k es una constante que depende de K y $c_{J,K} \in [a, b]$.

2.3.7. Método adaptativo de Simpson

Para reglas de cuadratura compuestas se utilizan intervalos pequeños de longitud h uniformemente en todo el intervalo de integración, para garantizar la precisión general. Sin embargo, esto no tiene en cuenta que algunas partes de la curva pueden tener grandes variaciones que requieren más

atención que otras. Es útil introducir un método que ajuste el tamaño de de la longitud de los intervalos de la partición del intervalo de integración, teniendo en cuenta los subintervalos donde la función a integrar tenga mayor variación. Esta técnica se llama cuadratura adaptativa y se basa en la regla de Simpson, que utiliza dos subintervalos sobre el subintervalo genérico $[a_k, b_k]$:

$$S(a_k, b_k) = \frac{h}{3}(f(a_k) + 4f(c_k) + f(b_k)), \tag{2.53}$$

donde $c_k = \frac{1}{2}(a_k + b_k)$ es el punto medio del intervalo $[a_k, b_k]$ y $h = (b_k - a_k)/2$. Además si $f \in C^4[a_k, b_k]$ y existe un punto $d_1 \in [a_k, b_k]$ tal que:

$$\int_{a_k}^{b_k} f(x)\mathrm{d}x = S(a_k, b_k) - h^5 \frac{f^{(4)}(d_1)}{90}. \tag{2.54}$$

Para realizar una regla compuesta de Simpson de mayor precisión se pueden utilizar cuatro subintervalos de $[a_k, b_k]$ mediante la subdivisión de este intervalo en dos subintervalos iguales $[a_{k1}, b_{k1}]$ y $[a_{k2}, b_{k2}]$ y aplicando la fórmula (2.53) recursivamente sobre cada uno de los subintervalos nuevos. Solo se necesitan dos evaluaciones adicionales de $f(x)$ y el resultado es:

$$S(a_{k1}, b_{k1}) + S(a_{k2}, b_{k2}) = \frac{h}{6}(f(a_{k1}) + 4f(c_{k1}) + f(b_{k1})) + \frac{h}{6}(f(a_{k2}) + 4f(c_{k2}) + f(b_{k2})), \tag{2.55}$$

donde $a_{k1} = a_k, b_{k1} = a_{k2} = c_k, b_{k2} = b_k$, c_{k1} es el punto medio de $[a_{k1}, b_{k1}]$ y c_{k2} es el punto medio de $[a_{k2}, b_{k2}]$. En (2.55) la longitud de los subintervalos es $h/2$ por lo que tenemos los factores $h/6$ en su parte de la derecha. Además si $f \in C^4[a_k, b_k]$, entonces existe un punto $d_2 \in [a_k, b_k]$ tal que:

$$\int_{a_k}^{b_k} f(x)\mathrm{d}x = S(a_{k1}, b_{k1}) + S(a_{k2}, b_{k2}) - \frac{h^5}{16}\frac{f^{(4)}(d_2)}{90}. \tag{2.56}$$

Asumiendo que $f^{(4)}(d_1) \approx f^{(4)}(d_2)$ obtenemos:

$$-h^5 \frac{f^{(4)}(d_2)}{90} \approx \frac{16}{15}(S(a_{k1}, b_{k1}) + S(a_{k2}, b_{k2}) - S(a_k, b_k)). \tag{2.57}$$

Como resultado final obtenemos:

$$\left| \int_{a_k}^{b_k} f(x)\mathrm{d}x - S(a_{k1}, b_{k1}) - S(a_{k2}, b_{k2}) \right| \approx \frac{1}{15} \left| S(a_{k1}, b_{k1}) + S(a_{k2}, b_{k2}) - S(a_k, b_k)) \right|. \tag{2.58}$$

2.3.7.1. Prueba de Precisión

Supongamos que se especifica una tolerancia $\epsilon_k > 0$ para el intervalo $[a_k, b_k]$. Entonces, si se tiene que

$$\frac{1}{10} \left| (S(a_{k1}, b_{k1}) + S(a_{k2}, b_{k2}) - S(a_k, b_k)) \right| < \epsilon_k, \tag{2.59}$$

entonces

$$\left| \int_{a_k}^{b_k} f(x)\mathrm{d}x - S(a_{k1}, b_{k1}) - S(a_{k2}, b_{k2}) \right| < \epsilon_k. \tag{2.60}$$

La regla compuesta de Simpson se utiliza para aproximar la integral:

$$\int_{a_k}^{b_k} f(x)\mathrm{d}x \approx (S(a_{k1}, b_{k1}) + S(a_{k2}, b_{k2}) \tag{2.61}$$

y la cota de error para esta aproximación sobre $[a_k, b_k]$ es ϵ_k.

La técnica de cuadratura adaptativa se basa en el uso de las reglas de Simpson y se inicia con una terna $\{[a_0, b_0], \epsilon_0\}$, donde $[a_0, b_0]$ es un intervalo y ϵ_0 es la tolerancia establecida para la cuadratura numérica sobre $[a_0, b_0]$. Este intervalo se divide en dos subintervalos etiquetados como $[a_{01}, b_{01}]$ y $[a_{02}, b_{02}]$. Si se supera la prueba de precisión (2.59), se aplica la fórmula de cuadratura a $[a_0, b_0]$ y finalizamos. Si la prueba en (2.59) no se cumple, los subintervalos se reetiquetan como $[a_1, b_1]$ y $[a_2, b_2]$ sobre los que usaremos tolerancias $\epsilon_1 = \frac{1}{2}\epsilon_0$ y $\epsilon_2 = \frac{1}{2}\epsilon_0$, respectivamente. De esta forma, tenemos dos intervalos con sus tolerancias asociadas, es decir, las ternas $\{[a_1, b_1], \epsilon_1\}$ y $\{[a_2, b_2], \epsilon_2\}$, que debemos considerar para continuar con el método.

En la siguiente iteración, considera primero la terna $\{[a_1, b_1], \epsilon_1)\}$ y se divide el intervalo $[a_1, b_1]$ en dos subintervalos $[a_{11}, b_{11}]$ y $[a_{12}, b_{12}]$. Si ambos subintervalos cumplen la prueba de precisión (2.59) con tolerancia ϵ_1, se aplica la fórmula de cuadratura a $[a_1, b_1]$ y acabamos. Si falla la prueba con tolerancia ϵ_1, cada subintervalo debe ser refinado y probado en la siguiente iteración con una tolerancia reducida $\frac{1}{2}\epsilon_1$. Además, en el segundo paso también se debe considerar la terna $\{[a_2, b_2], \epsilon_2)\}$ y se divide el intervalo $[a_2, b_2]$ en $[a_{21}, b_{21}]$ y $[a_{22}, b_{22}]$. Si ambos subintervalos cumplen la prueba de precisión (2.59) con tolerancia ϵ_2, se aplica la fórmula de cuadratura a $[a_2, b_2]$ y acabamos. Si falla la prueba en (2.59) con tolerancia ϵ_2, cada subintervalo debe ser refinado y probado en la siguiente iteración.

La regla de Simpson reduce el error en aproximadamente un factor de 16 cada vez que se divide el intervalo de integración en subintervalos más pequeños y se aplica la regla compuesta de Simpson a cada subintervalo. De esta forma, el proceso finalizará tras un número finito de pasos.

Capítulo 3

Código

En este capítulo se realizará una explicación de los códigos implementados en Python para la realización de los distintos métodos de integración numérica presentados en el anterior capítulo.

3.1. Librerías utilizadas

```
import numpy as np
from sympy import sin as sin
from sympy import cos as cos
from sympy import sqrt as sqrt
from sympy import log as log
from sympy import pi as pi
from sympy import E as E
from sympy import tan as tan
import sympy as sp

x = sp.symbols('x')
```

En el código proporcionado, se importan las siguientes librerías y se asignan alias a algunas funciones:

1. **numpy** (importado como *np*): Es una biblioteca de Python utilizada para realizar cálculos numéricos y trabajar con arreglos multidimensionales (véase [4]). Se utiliza para realizar operaciones matemáticas eficientes en matrices y vectores.
2. **sympy**: Es una biblioteca de matemáticas simbólicas en Python (véase

[5, 6]). Proporciona herramientas para trabajar con expresiones matemáticas simbólicas, realizar cálculos simbólicos, resolver ecuaciones, derivar e integrar, entre otros.

3. **sin, cos, sqrt, log, pi, E, tan**: Son funciones específicas importadas desde la biblioteca *SymPy*. Estas funciones permiten trabajar con operaciones matemáticas simbólicas, como el seno, coseno, raíz cuadrada, logaritmo, pi, número de Euler y tangente.

4. **sp** (alias para *SymPy*): Se utiliza para acceder a las funciones y características de la biblioteca SymPy (véase [5, 6]) mediante el alias *sp*.

Además, se crea un símbolo matemático x utilizando la función *symbols* de *SymPy*, que se utilizará posteriormente en construcciones simbólicas y cálculos matemáticos.

3.2. Fórmula de los Rectángulos Compuesta

```
def rectangulosCompuesta(f,a,b,n):

    pasos=np.zeros((11,4))
    h = (b-a)/float(n)
    suma = 0
    for i in range(n):
        suma += f.subs(x, a+h*i)
        if(i<=9):
            pasos[i,:]=(i+1,a+h*i,float(f.subs(x,
             ↪  a+h*i)),suma)
        if(n>10 and i==(n-1)):
            pasos[10,:]=(i+1,a+h*i,float(f.subs(x,
             ↪  a+h*i)),suma)
    # Construir la tabla en latex
    res = "\\left[ \\begin{array}{ccccc}iter & punto & valor
     ↪  & suma \\ \hline"
    for paso in pasos:
        res += str(paso[0]) + "&" + str(paso[1]) + "&" +
         ↪  str(paso[2]) + "&" + str(paso[3]) +"\\\\ "
    res += "\\end{array}\\right]"
    resFinal=str(float(suma*h))
    resH=str(h)
    # Retornar el resultado
```

Código

```
return { "pasos": [{ "descripcion": "Esta es la longitud
 ↪  del intervalo", "pasoLatex": resH },
        { "descripcion": "Esta es la taba de
         ↪  iteraciones", "pasoLatex": res },
                  {"descripcion": "Multiplicamos suma
                   ↪  por h y obtenemos el resultado
                   ↪  final","pasoLatex":resFinal}] }
```

El código proporcionado define una función llamada **rectangulosCompuesta** que realiza la integración numérica utilizando el método de rectángulos compuesta. A continuación, se explica paso a paso lo que hace este código:

1. **def rectangulosCompuesta(f, a, b, n)**: define la función **rectangulosCompuesta** con cuatro parámetros: f (la función a integrar), a y b (los límites inferior y superior de integración) y n (el número de subintervalos).

2. **pasos = np.zeros((11, 4))**: crea una matriz de ceros de dimensiones 11x4 utilizando la biblioteca NumPy. Esta matriz se utiliza para almacenar los pasos intermedios de la integración.

3. **h = (b - a) / float(n)**: calcula el tamaño de cada subintervalo dividiendo la diferencia entre los límites b y a por el número de subintervalos n. La función `float()` se utiliza para asegurar que la división se realice como una división de punto flotante.

4. **suma = 0**: inicializa la variable `suma` en cero, que se utilizará para acumular las sumas de las evaluaciones de la función en los puntos intermedios.

5. **for i in range(n)**: inicia un bucle `for` que se repetirá n veces, es decir, una vez para cada subintervalo.

6. **suma += f.subs(x, a + h * i)**: acumula el valor de la evaluación de la función f en el punto medio de cada subintervalo. El método `.subs()` se utiliza para sustituir el símbolo x en la función f por el valor correspondiente.

7. **if (i <= 9)**: verifica si el índice i es menor o igual a 9, es decir, si es uno de los primeros 10 subintervalos.

8. **pasos[i, :] = (i, a + h * i, float(f.subs(x, a + h * i)), suma)**: guarda los pasos intermedios en la matriz `pasos`. En la fila i, se almacena el índice i, el punto medio del subintervalo, el valor de la función evaluada en ese punto medio y la suma acumulada hasta ese momento.

9. **if (i == (n - 1))**: verifica si el índice i es igual al número total de subintervalos menos uno, es decir, si es el último subintervalo.
10. **pasos[10, :] = (10, a + h * i, float(f.subs(x, a + h * i)) , suma)**: guarda el último paso intermedio en la matriz `pasos`. En la última fila, se almacena el índice 10, el punto medio del último subintervalo, el valor de la función evaluada en ese punto medio y la suma acumulada hasta ese momento.
11. La parte final del proceso construye todos los resultados que hemos obtenido como la longitud del intervalo, la matriz de pasos y el resultado y los devuelve en formato tabla para que así sea más visual este apartado en la aplicación didáctica.

3.3. Fórmula del Punto Medio Compuesta

```
def puntoMedioCompuesta(f,a, b, n):

    pasos=np.zeros((11,4))
    h = (b-a)/float(n)
    j = a
    suma =0
    for i in range(n):
        suma += f.subs(x,(j+h+j)/2)
        if(i<=9):
            pasos[i,:]=(i+1,(j+h+j)/2,float(f.subs(x,(j+h+j)/2)),suma)
        if( n>10 and i==(n-1)):
            pasos[10,:]=(i+1,(j+h+j)/2,float(f.subs(x,(j+h+j)/2)),suma)
        j += h

    res = "\\left[ \\begin{array}{ccccc}iter & punto & valor &
    suma \\ \hline"
    for paso in pasos:
        res += str(paso[0]) + "&" + str(paso[1]) + "&" + str(paso[2])
        + "&" + str(paso[3]) +"\\\\ "
    res += "\\end{array}\\right]"
    resFinal=str(float(suma*h))
    resH=str(h)
    # Retornar el resultado
    return { "pasos": [{ "descripcion": "Esta  es la longitud
    del intervalo", "pasoLatex": resH },
```

```
{ "descripcion": "Esta es la tabla de iteraciones"
, "pasoLatex": res },
{"descripcion": "Multiplicamos suma por
h y obtenemos el resultado final","pasoLatex":resFinal}] }
```

1. **def puntoMedioCompuesta(f, a, b, n)**: define la función **puntoMedioCompuesta** con cuatro parámetros: `f` (la función a evaluar), `a` y `b` (los límites inferior y superior del intervalo) y `n` (el número de subintervalos).
2. **pasos = np.zeros((11, 4))**: inicializa una matriz de ceros de tamaño 11×4 utilizando la biblioteca NumPy. Esta matriz se utiliza para almacenar los pasos intermedios de la evaluación de la función en cada subintervalo.
3. **h = (b-a)/float(n)**: calcula el tamaño de cada subintervalo dividiendo la diferencia entre `b` y `a` por `n`.
4. **j = a**: inicializa la variable `j` con el valor de `a`. Esta variable se utiliza para iterar sobre los subintervalos.
5. **suma = 0**: inicializa la variable `suma` en cero. Esta variable se utiliza para acumular la suma de las evaluaciones de la función en cada punto medio de los subintervalos.
6. **for i in range(n)**: inicia un bucle `for` que itera `n` veces, correspondiendo al número de subintervalos.
7. **suma += f.subs(x, (j+h+j)/2)**: acumula el valor de la evaluación de la función en el punto medio del subintervalo utilizando la expresión `(j+h+j)/2`. El resultado se suma a la variable `suma`.
8. **if(i <= 9)**: verifica si el índice `i` es menor o igual a 9. Si se cumple esta condición, se asignan los valores correspondientes a la matriz de pasos en la fila `i`.
9. **pasos[i,:]=(i+1,(j+h+j)/2,float(f.subs(x,(j+h+j)/2)),suma))**: asigna los valores a la matriz de pasos en la fila `i`. Los valores son el índice `i`, el límite inferior del subintervalo, el valor de la función evaluada en el punto medio del subintervalo, y la `suma` hasta ese momento.
10. **pasos[10,:]=(i+1,(j+h+j)/2,float(f.subs(x,(j+h+j)/2)),suma))**: asigna los valores a la matriz de pasos en la última fila. Los valores son el índice `10`, el límite inferior del último subintervalo, el valor de la función evaluada en el punto medio del último subintervalo y la `suma` hasta ese momento.

11. **j += h**: actualiza el valor de `j` sumándole `h`, para avanzar al siguiente subintervalo.
12. La parte final del proceso construye todos los resultados que hemos obtenido como la longitud del intervalo, la matriz de pasos y el resultado y los devuelve en formato tabla para que así sea más visual este apartado en la aplicación didáctica.

En resumen, la función **puntoMedioCompuesta** implementa la regla del punto medio compuesta para aproximar la integral de una función en un intervalo dado utilizando un número específico de subintervalos. Calcula la suma acumulada de las evaluaciones de la función en los puntos medios de los subintervalos, y guarda los pasos intermedios en una matriz. Al final, devuelve el resultado de la suma acumulada y los pasos intermedios en un diccionario en formato tabla para que sea más visual.

3.4. Fórmula de los Trapecios Compuesta

```
def trapecioCompuesta(f,a, b, n):

    pasos=np.zeros((11,4))
    h = (b - a) / float(n)
    suma = 0
    for i in range(1, n-1):
        suma += f.subs(x,a + i * h)
        if(i<=9):
            pasos[i,:]=(i+1,a+h*i,f.subs(x, a+h*i),suma)
        if(n>10 and i==(n-1)):
            pasos[10,:]=(i+1,a+h*i,f.subs(x, a+h*i),suma)
    resFinal=str(float(h/2 * (f.subs(x,a) + f.subs(x,b))  +
     ↪  h*suma))

    res = "\\left[ \\begin{array}{ccccc}iter & punto & valor
     ↪  & suma \\ \hline"
    for paso in pasos:
        res += str(paso[0]) + "&" + str(paso[1]) + "&" +
         ↪  str(paso[2]) + "&" + str(paso[3]) +"\\\\ "
    res += "\\end{array}\\right]"
    resH=str(h)
    # Retornar el resultado
```

Código

```
return { "pasos": [{ "descripcion": "Esta es la longitud
 ↪  del intervalo", "pasoLatex": resH },
        { "descripcion": "Esta es la taba de
         ↪  iteraciones", "pasoLatex": res },
 {"descripcion": "Multiplicamos suma por h y lo sumamos a
  ↪  la fo´rmula dle tarpecio en los extremos. Obtenemos
  ↪  el resultado final:","pasoLatex":resFinal}] }
```

1. **def trapecioCompuesta(f, a, b, n)**: define una función llamada **trapecioCompuesta** que toma cuatro parámetros: `f` (la función a integrar), `a` y `b` (los límites inferior y superior de integración) y `n` (el número de subintervalos).
2. **pasos=np.zeros((11,4))**: crea una matriz de ceros de dimensiones 11x4 utilizando la biblioteca NumPy. Esta matriz se utilizará para almacenar los pasos intermedios de la integración.
3. **h = (b - a) / float(n)**: calcula el tamaño de cada subintervalo dividiendo la diferencia entre los límites `b` y `a` por el número de subintervalos `n`. La función `float()` se utiliza para asegurar que la división se realice como una división de punto flotante.
4. **suma = 0**: inicializa la variable `suma` en cero, que se utilizará para acumular las sumas de las evaluaciones de la función en los puntos intermedios.
5. **for i in range(1, n-1)**: inicia un bucle `for` que se repetirá `n-2` veces, es decir, una vez para cada subintervalo, excepto el primero y el último.
6. **suma += f.subs(x, a + i * h)**: acumula el valor de la evaluación de la función `f` en el punto medio de cada subintervalo. El método `.subs()` se utiliza para sustituir el símbolo `x` en la función `f` por el valor correspondiente, que se calcula como `a + i * h`.
7. **if(i <= 9)**: verifica si el índice `i` es menor o igual a 9, es decir, si es uno de los primeros 10 subintervalos.
8. **pasos[i, :] = (i, a + h * i, float(f.subs(x, a + h * i)) , suma)**: guarda los pasos intermedios en la matriz `pasos`. En la fila `i`, se almacena el índice `i`, el punto medio del subintervalo calculado como `a + h * i`, el valor de la función evaluada en ese punto medio y la suma acumulada hasta ese momento.
9. **if(i == (n-1))**: verifica si el índice `i` es igual a `n-1`, es decir, si es el último subintervalo.

10. **pasos[10, :] = (10, a + h * i, float(f.subs(x, a + h * i)), suma)**: guarda el último paso intermedio en la fila 10 de la matriz `pasos`. En esta fila se almacena el índice 10, el punto medio del último subintervalo calculado como `a + h * i`, el valor de la función evaluada en ese punto medio, y la suma acumulada hasta ese momento.

    El resultado de la integración se calcula como `h/2 * (f.subs(x, a) + f.subs(x, b)) + h * suma`. Representa el área aproximada bajo la curva utilizando la fórmula del trapecio compuesta.

    - El valor de `h/2 * (f.subs(x, a) + f.subs(x, b))` representa el área del trapecio formado por los puntos `a` y `b`.
    - `h * suma` corresponde a la suma acumulada de las evaluaciones de la función en los puntos intermedios multiplicada por `h`, que representa el área de los trapecios adicionales en los subintervalos internos.

11. La parte final del proceso construye todos los resultados que hemos obtenido como la longitud del intervalo, la matriz de pasos y el resultado y los devuelve en formato tabla para que así sea más visual este apartado en la aplicación didáctica.

En resumen, la funcion **trapecioCompuesta** implementa el método de integración numérica del trapecio compuesta. Toma una función `f`, límites de integración `a` y `b`, y el número de subintervalos `n`. Calcula el tamaño del subintervalo `h`, acumula las sumas de las evaluaciones de la función en los puntos intermedios y guarda los pasos intermedios en una matriz. Finalmente, retorna el resultado de la integración y la matriz de pasos en un diccionario en formato tabla.

3.5. Fórmula de Simpson Compuesta

```
def simpsonCompuesta(f,a, b, n):
    """
    Integración numérica de la función f en el intervalo [a,
 ↪  b]
    usando la fórmula de Simpson compuesta con n
 ↪  subintervalos.
    """

    pasos=np.zeros((11,7))
```

Código

```
h = (b - a) / float(2*n)
s1=0
s2=0

for i in range(1,n):
    s1 += f.subs(x,a+h*(2*i-1))
    if(i<=9):
        pasos[i,0]=i+1
        pasos[i,1]=a+h*(2*i-1)
        pasos[i,2]=f.subs(x,a+h*(2*i-1))
        pasos[i,3]=s1
    if(n>10 and i==(n-1)):
        pasos[10,0]=i+1
        pasos[10,1]=a+h*(2*i-1)
        pasos[10,2]=f.subs(x,a+h*(2*i-1))
        pasos[10,3]=s1
for i in range(1,n-1):
    s2 += f.subs(x,a+h*2*i)
    if(i<=9):
        pasos[i,4]=a+h*2*i
        pasos[i,5]=f.subs(x,a+h*2*i)
        pasos[i,6]=s2
    if(n>10 and i==(n-2)):
        pasos[10,4]=a+h*2*i
        pasos[10,5]=f.subs(x,a+h*2*i)
        pasos[10,6]=s2

 ↪  resFinal=str(float(h*(f.subs(x,a)+f.subs(x,b)+4*s1+2*s2)/3))

res = "\\left[ \\begin{array}{ccccc}iter & punto1 &
 ↪  valor1 & suma1 & punto2 & valor2 & suma2 \\ \hline"
for paso in pasos:
    res += str(paso[0]) + "&" + str(paso[1]) + "&" +
     ↪  str(paso[2]) + "&" + str(paso[3]) + "&" +
     ↪  str(paso[4]) + "&" + str(paso[5]) + "&" +
     ↪  str(paso[6]) +"\\\\ "
res += "\\end{array}\\right]"
resH=str(h)
# Retornar el resultado
return { "pasos": [{ "descripcion": "Esta es la longitud
 ↪  del intervalo", "pasoLatex": resH },
```

Código

```
{ "descripcion": "Esta es la tabla de
↪  iteraciones", "pasoLatex": res },
            {"descripcion": "Aplicando la formula
             ↪  del trapecio, obtenemos el
             ↪  resultado
             ↪  final:","pasoLatex":resFinal}] }
```

1. **def simpsonCompuesta(f,a, b, n)**: define la función **simpsonCompuesta** con cuatro parámetros: `f` (la función a integrar), `a` y `b` (los límites inferior y superior de integración) y `n` (el número de subintervalos). Esta función realiza la integración numérica de la función `f` en el intervalo `[a, b]` utilizando la fórmula de Simpson compuesta con `n` subintervalos.
2. **pasos=np.zeros((11,7))**: crea una matriz de ceros de dimensiones 11x7 utilizando la biblioteca NumPy. Esta matriz se utilizará para almacenar los pasos intermedios de la integración.
3. **h = (b - a) / float(2n)**: calcula el tamaño de cada subintervalo dividiendo la diferencia entre los límites `b` y `a` por el doble del número de subintervalos `n`. La función `float()` se utiliza para asegurar que la división se realice como una división de punto flotante.
4. **s1=0 s2=0**: inicializa la variable `s1 s2` en cero, que se utilizará para acumular las sumas de las evaluaciones de la función en los puntos impares y pares de los subintervalos.
5. **for i in range(1,n)**: inicia un bucle `for` que se repetirá `n-1` veces, es decir, una vez para cada subintervalo, excepto el último.
6. **s1 += f.subs(x,a+h(2i-1))**: acumula el valor de la evaluación de la función `f` en el punto impar de cada subintervalo. El método `.subs()` se utiliza para sustituir el símbolo `x` en la función `f` por el valor correspondiente.
7. **if(i<=9)**: verifica si el índice `i` es menor o igual a 9, es decir, si es uno de los primeros 10 subintervalos.
8. **pasos[i,0]=i**: guarda el índice `i` en la primera columna de la matriz `pasos`.
9. **pasos[i,1]=a+h*(2i-1)**: guarda el punto medio del subintervalo en la segunda columna de la matriz `pasos`.
10. **pasos[i,2]=f.subs(x,a+h(2i-1))**: guarda el valor de la función evaluada en el punto medio del subintervalo en la tercera columna de la matriz

`pasos`.

11. **pasos[i,3]=s1**: guarda la suma acumulada hasta ese momento en la cuarta columna de la matriz `pasos`.
12. **if(i==(n-1))**: verifica si el índice `i` es igual a `n-1`, es decir, si es el último subintervalo.
13. **pasos[10,0]=i**: guarda el índice `i` en la primera columna de la última fila de la matriz `pasos` como en los 9 primeros subintervalos.
14. **for i in range(1,n-1)**: inicia un bucle `for` que se repetirá `n-2` veces, es decir, una vez para cada subintervalo, excepto el primero y el último.
15. **s2 += f.subs(x,a+h2i)**: acumula el valor de la evaluación de la función `f` en el punto par de cada subintervalo.
16. **if(i<=9)**: verifica si el índice `i` es menor o igual a 9, es decir, si es uno de los primeros 10 subintervalos y realiza el guardado de los pasos como se explicó anteriormente.
17. **if(i==(n-1))**: verifica si el índice `i` es igual a `n-1`, es decir, si es el último subintervalo par y guarda en la matriz los pasos correspondientes.
18. La parte final del proceso construye todos los resultados que hemos obtenido como la longitud del intervalo, la matriz de pasos y el resultado y los devuelve en formato tabla para que así sea más visual este apartado en la aplicación didáctica.

En resumen, la función **simpsonCompuesta** implementa la fórmula de Simpson compuesta para realizar la integración numérica de una función en un intervalo dado. Utiliza una matriz para almacenar los pasos intermedios de la integración, calculando las sumas parciales en los puntos impares y pares de los subintervalos. Luego, aplica la fórmula de Simpson compuesta para obtener el resultado de la integración y devuelve tanto el resultado como los pasos intermedios en un diccionario en formato tabla.

3.6. Fórmula de los Trapecios Recursiva

```
def trapecioRecursivo(f,a, b, n):

    M=1
    h=b-a
    T=np.zeros(n+1)
```

Código

```
    pasos=np.zeros((10,2))
    T[1]=h*(f.subs(x,a)+f.subs(x,b))/2
    for j in range(1,n):
        M=2*M
        h=h/float(2)
        s=0
        for k in range(1,M//2+1):
            s+=f.subs(x,a+h*(2*k-1))
        T[j+1]=T[j]/2+h*s
        if(j<=9):
            pasos[j,:]=(j+1,T[j])
        if(j>10 and j==(n-1)):
            pasos[10,:]=(j+1,T[j])
    res = "\\left[ \\begin{array}{ccccc}iter & valor \\ \hline"
    for paso in pasos:
        res += str(paso[0]) + "&" + str(paso[1])  +"\\\\ "
    res += "\\end{array}\\right]"

    # Retornar el resultado
    return { "pasos":[
            { "descripcion": "Esta es la tabla de iteraciones",
            "pasoLatex": res }] }
```

1. **def trapecioRecursivo(f, a, b, n)**: Define la función **trapecioRecursivo** con cuatro parámetros: `f` (la función a integrar), `a` y `b` (los límites inferior y superior de integración) y `n` (el número de subdivisiones para el método del trapecio).
2. **M = 1**: Inicializa la variable `M` con el valor 1. Esta variable se utiliza para realizar los cálculos recursivos en cada iteración.
3. **h = b - a**: Calcula el tamaño inicial del intervalo de integración y lo almacena en la variable `h`.
4. **T = np.zeros(n+1)**: Crea un arreglo de ceros de longitud `n+1` utilizando la biblioteca NumPy. Este arreglo se utiliza para almacenar los valores de las aproximaciones de la integral en cada iteración.
5. **pasos = np.zeros()**: Crea una matriz vacía. Esta matriz se utilizaría para almacenar información sobre los pasos intermedios del cálculo, pero en el código proporcionado no se ha especificado la dimensión.
6. **T[1] = h * (f.subs(x, a) + f.subs(x, b)) / 2**: Calcula y almacena en `T[1]` la primera aproximación de la integral utilizando la fórmula del trapecio

con los límites `a` y `b`.

7. **for j in range(1, n)**: Inicia un bucle `for` que se repetirá `n-1` veces, es decir, una vez para cada iteración del método del trapecio recursivo.
8. **M = 2 * M**: Actualiza el valor de `M` multiplicándolo por 2 en cada iteración. Esto duplica el número de subintervalos en cada paso recursivo.
9. **h = h / float(2)**: Actualiza el tamaño del subintervalo `h` dividiéndolo por 2 en cada iteración.
10. **s = 0**: Inicializa la variable `s` en cero, que se utilizará para acumular las sumas de las evaluaciones de la función en los puntos intermedios de cada subintervalo.
11. **for k in range(1, M//2 + 1)**: Inicia un bucle `for` que se repetirá `M//2` veces. Este bucle se utiliza para calcular la suma de las evaluaciones de la función en los puntos intermedios de cada subintervalo.
12. **s += f.subs(x, a + h * (2 * k - 1))**: Acumula el valor de la evaluación de la función `f` en el punto medio de cada subintervalo en la variable `s`. Se utiliza la fórmula `a + h * (2 * k - 1)` para obtener el punto medio de cada subintervalo.
13. **T[j+1] = T[j] / 2 + h * s**: Calcula y almacena en `T[j+1]` la aproximación de la integral en la iteración actual. Se utiliza la fórmula recursiva del método del trapecio para obtener esta aproximación, utilizando el valor anterior `T[j]`, la suma acumulada `s` y el tamaño del subintervalo `h`.
14. La parte final del proceso construye todos los resultados que hemos obtenido como el número de iteración y el vector de valores y los devuelve en formato tabla para que así sea más visual este apartado en la aplicación didáctica.

    En resumen, este código implementa el método **trapecioRecursivo** para realizar la integración numérica de una función `f` en un intervalo dado `[a, b]`, utilizando `n` subintervalos. Calcula los términos de la fórmula del trapecio de forma recursiva, actualizando la longitud de los subintervalos y acumulando las evaluaciones de la función en los puntos medios. Devuelve el resultado de la integración y la lista de todos los términos calculados durante el proceso en formato tabla.

3.7. Método de Romberg

```
def romberg( f,a, b, n, tol):
    M = 1
    h = (b - a)
    err = 1
    J = 0
    R = np.zeros((6,6),float)
    R[0, 0] = h * (f.subs(x,a) + f.subs(x,b)) / float(2)
    while ((err > tol and J < n) or J < n):
        J += 1
        h =h/float(2)
        s = 0

        for p in range(1, M):
            s += f.subs(x,a + (2*p - 1) * h)

        if J + 1 >= R.shape[0] or J + 2 >= R.shape[1]:
           new_R = np.zeros((R.shape[0] + 1, R.shape[1] + 1))
           new_R[:R.shape[0], :R.shape[1]] = R
           R = new_R

        R[J, 0] = R[J-1, 0] / float(2) + h * s
        M *= 2
        for k in range(1, J):
            R[J, k] = R[J, k-1] + (R[J, k-1] - R[J-1, k-1]) /
             ↪  float((4**k-1))
        err = abs(R[J, J] - R[J-1, J-1])
    quad = R[n, n-1]

    resFinal=str(quad)
    resErr=str(err)
    resH=str(h)
    res = "\\left[ \\begin{array}{ccccc}"
    for paso in R:
        res += str(paso[0]) + "&" + str(paso[1]) + "&" +
         ↪  str(paso[2]) + "&" + str(paso[3]) +"&" +
         ↪  str(paso[4]) +"&" + str(paso[5]) +"\\\\ "
    res += "\\end{array}\\right]"

    # Retornar el resultado
```

Código

```
return { "pasos": [
        { "descripcion": "Esta es la tabla de Romberg",
        ↪  "pasoLatex": res },
                {"descripcion": "Obtenemos el
                ↪  resultado
                ↪  final:","pasoLatex":resFinal},
                {"descripcion": "Obtenemos el
                ↪  error:","pasoLatex":resErr},
                {"descripcion": "Obtenemos la longitud
                ↪  del menor intervalo
                ↪  usado:","pasoLatex":resH}] }
```

1. **def romberg(f,a, b, n, tol):**: Define la función **romberg** con cinco parámetros: `f` (la función a integrar), `a` y `b` (los límites inferior y superior de integración), `n` (el número de subdivisiones para el método del trapecio) y `tol` la tolerancia al error que tendrá el método.
2. **M = 1**: Se inicializa la variable `M` con el valor 1. Esta variable se utiliza para realizar un seguimiento de los subintervalos utilizados en el método de Romberg.
3. **h = (b - a)**: Se calcula la longitud del intervalo de integración y se asigna a la variable `h`.
4. **err = 1**: Se inicializa la variable `err` con el valor 1. Esta variable se utiliza para controlar el criterio de convergencia del método de Romberg.
5. **J = 0**: Se inicializa la variable `J` con el valor 0. Esta variable se utiliza para realizar un seguimiento del nivel actual en el proceso de refinamiento del método de Romberg.
6. **R = np.zeros((6,6),float)**: Se crea una matriz de ceros de tamaño 6x6 y tipo de datos float. Esta matriz se utiliza para almacenar los resultados intermedios del método de Romberg.
7. **R[0, 0] = h * (f.subs(x,a) + f.subs(x,b)) / float(2)**: Se calcula y se asigna el valor inicial de la matriz `R`. Este valor corresponde a la aproximación inicial utilizando el método del trapecio compuesto.
8. **while ((err >tol and J <n) or J <n):**: Se inicia un bucle `while` que continuará hasta que se cumpla el criterio de convergencia o se alcance el número máximo de refinamientos. El bucle se ejecuta al menos una vez.
9. **J += 1**: Se incrementa en 1 el valor de `J` en cada iteración del bucle.

Esto indica el nivel actual de refinamiento en el método de Romberg.

10. **h = h/float(2)**: Se reduce a la mitad el valor de `h` en cada iteración del bucle. Esto implica dividir el intervalo en subintervalos más pequeños en cada nivel de refinamiento.

11. **s = 0**: Se inicializa la variable `s` con el valor 0. Esta variable se utiliza para calcular la suma de los términos necesarios para la aproximación actual en el método de Romberg.

12. **for p in range(1, M)**: Se inicia un bucle `for` que se repite `M-1` veces. Este bucle se utiliza para calcular la suma de los términos necesarios para la aproximación actual en el método de Romberg.

13. **s += f.subs(x,a + (2p - 1) * h)**: Se suma al valor acumulado `s` el valor de la función evaluada en el punto medio de cada subintervalo. Esto se realiza mediante la expresión `a + (2p - 1) * h`, que calcula el punto medio correspondiente al subintervalo actual.

14. **if J + 1 >= R.shape[0] or J + 2 >= R.shape[1]**: Se verifica si es necesario ampliar la matriz `R` para almacenar más resultados intermedios. Esto ocurre cuando se alcanza el tamaño máximo predefinido de la matriz.

15. **newR = np.zeros((R.shape[0] + 1, R.shape[1] + 1))**: Se crea una nueva matriz `newR` con un tamaño ampliado en una fila y una columna respecto a la matriz `R` actual.

16. **newR[:R.shape[0], :R.shape[1]] = R**: Se copian los elementos de la matriz `R` actual a la nueva matriz `newR`, preservando los valores existentes.

17. **R = newR**: Se actualiza la referencia de la matriz `R` para apuntar a la nueva matriz ampliada.

18. **R[J, 0] = R[J-1, 0] / float(2) + h * s**: Se calcula y se asigna el valor de la aproximación actual en la matriz `R`. Esta aproximación se obtiene utilizando la extrapolación de Richardson.

19. **M *= 2**: Se duplica el valor de `M` en cada iteración del bucle. Esto implica que se utilizan el doble de subintervalos en cada nivel de refinamiento.

20. **for k in range(1, J)**: Se inicia un bucle `for` que se repite `J-1` veces. Este bucle se utiliza para realizar la extrapolación de Richardson y mejorar la precisión de la aproximación.

21. **R[J, k] = R[J, k-1] + (R[J, k-1] - R[J-1, k-1]) / float((4**k-1))**: Se realiza la extrapolación de Richardson para calcular y asignar el valor de la aproximación en la matriz `R`.

22. **err = abs(R[J, J] - R[J-1, J-1])**: Se calcula el error entre la aproximación actual y la aproximación anterior utilizando los elementos correspondientes de la matriz `R`.

23. **`quad = R[n, n-1]`**: Se obtiene la aproximación final del método de Romberg a partir de la matriz `R`. Esta aproximación se encuentra en la posición `R[n, n-1]`, donde `n` representa el nivel máximo de refinamiento.

24. La parte final del proceso construye todos los resultados que hemos obtenido como la longitud del intervalo, la matriz de pasos y el resultado y los devuelve en formato tabla para que así sea más visual este apartado en la aplicación didáctica.

En resumen, la función **romberg()** implementa el método de Romberg para aproximar la integral de una función `f` en un intervalo dado `[a, b]`. Utiliza la extrapolación de Richardson para mejorar la precisión de las aproximaciones y realiza un refinamiento progresivo dividiendo el intervalo en subintervalos más pequeños. El bucle principal se repite hasta que se cumpla el criterio de convergencia (el error es menor que la tolerancia) o se alcanza el número máximo de refinamientos. El resultado final se devuelve en forma de diccionario, que contiene la aproximación y la lista de pasos intermedios realizados en fomato tabla.

3.8. Método Adaptativo de Simpson

```
def srule(f, a, b, tol):
    # Calcula la regla de Simpson para un intervalo dad
    h = (b - a) / float(2)
    puntoMedio=(a+b)/float(2)
    S = (h/float(3)) * (f.subs(x, a) +
    4 * f.subs(x, puntoMedio) + f.subs(x, b))
    S2 = S
    tol1=tol
    err = tol
    Z = [a, b, S, S2, err, tol1]
    return Z
```

1. **h = (b - a) / float(2)**: Se calcula la mitad del ancho del intervalo dividido por 2 y se guarda en la variable `h`. Esto representa la longitud de cada subintervalo en el método de la regla de Simpson.

2. **puntoMedio = (a + b) / float(2)**: Se calcula el punto medio del intervalo y se guarda en la variable `puntoMedio`. Este punto se utiliza para evaluar la función en el centro del intervalo en la fórmula de la regla de Simpson.
3. **S = (h / float(3)) * (f.subs(x, a) + 4 * f.subs(x, puntoMedio) + f.subs(x, b))**: Se calcula la aproximación de la integral utilizando la fórmula de la regla de Simpson compuesta para un solo intervalo. Se evalúa la función en los extremos y en el punto medio del intervalo y se aplica la ponderación correspondiente. El resultado se guarda en la variable `S`.
4. **S2 = S**: Se guarda el valor de `S` en la variable `S2`. Esta variable se utilizará posteriormente para comparar la estimación actual con la estimación anterior y calcular el error.
5. **tol1 = tol**: Se guarda el valor de la tolerancia `tol` en la variable `tol1`. Esta variable se utilizará para realizar comparaciones y determinar cuándo se cumple la condición de tolerancia.
6. **err = tol**: Se inicializa el valor del error `err` con el valor de la tolerancia `tol`. Esto asegura que el bucle se ejecute al menos una vez.
7. **Z = [a, b, S, S2, err, tol1]**: Se crea una lista `Z` que contiene los valores relevantes para el cálculo de la regla de Simpson. Los elementos de la lista son el límite inferior del intervalo (`a`), el límite superior del intervalo (`b`), la estimación actual de la integral (`S`), la estimación anterior de la integral (`S2`), el error (`err`) y la tolerancia (`tol1`). Esta lista se utiliza para realizar el seguimiento de los resultados en cada iteración del método.
8. **`return Z`**: Se devuelve la lista `Z`, que contiene los resultados de la regla de Simpson para el intervalo dado.

En resumen, la función **srule** calcula una aproximación de la integral utilizando la regla de Simpson y devuelve una lista con los resultados.

```
def simpsonAdaptativo(f, a, b, tol):

    SRmat = np.zeros((30, 6))
    ite = 0
    done = 1
    SRvec = np.zeros((1, 6))
    SRvec = srule(f, a, b, tol)
    SRmat[0, 0:6] = SRvec
    m = 1
```

Código

```
state = ite
while(state == ite ):
    n = m

    for j in range(n-1, -1, -1):
        p = j
        SR0vec = SRmat[p, :]
        err = SR0vec[4]
        tol = SR0vec[5]
        if(tol <= err):
            state = done
            SR1vec = SR0vec
            SR2vec = SR0vec
            a = SR0vec[0]
            b = SR0vec[1]
            c = (a + b) / float(2)
            err = SR0vec[4]
            tol2 = SR0vec[5] / float(2)
            SR1vec = srule(f, a, c, tol2)
            SR2vec = srule(f, c, b, tol2)
            err = abs(SR0vec[2] - SR1vec[2] - SR2vec[2])
             ↪  / float(10)
            if(err < tol):
                SRmat[p, :] = SR0vec
                SRmat[p, 3] = SR1vec[2] + SR2vec[2]
                SRmat[p, 4] = err
            else:
                SRmat[p+1:m+1, :] = SRmat[p:m, :]
                m = m + 1
                SRmat[p, :] = SR1vec
                SRmat[p+1, :] = SR2vec
                state = ite
quad = np.sum(SRmat[:, 3])
err = np.sum(SRmat[:, 4])
SRmat = SRmat[0:m, 0:6]

resFinal=str(quad)
resErr=str(float(err))

res = "\\left[ \\begin{array}{ccccc}a & b & S1 & S2 &
 ↪  error & tolerancia \\ \hline"
```

Código

```
    for paso in SRmat:
        res += str(paso[0]) + "&" + str(paso[1]) + "&" +
         ↪  str(paso[2]) + "&" + str(paso[3]) +"&" +
         ↪  str(paso[4]) +"&" + str(paso[5]) +"\\\\ "
    res += "\\end{array}\\right]"
    resFinal=str(float(quad))

    # Retornar el resultado
    return {"pasos":[
            { "descripcion": "Esta es la tabla de
             ↪  iteraciones", "pasoLatex": res },
                        {"descripcion": "Obtenemos el
                         ↪  resultado
                         ↪  final","pasoLatex":resFinal},
                        { "descripcion": "Esta es el error",
                         ↪  "pasoLatex": resErr }] }
```

1. **Inicialización de variables**:
   - `SRmat`: Se crea una matriz de tamaño `(30, 6)` inicializada con ceros para almacenar los resultados intermedios de la regla de Simpson adaptativa.
   - `ite` y `done`: Se inicializan las variables de estado `ite` y `done` con los valores 0 y 1, respectivamente.
   - `SRvec`: Se crea una matriz de tamaño `(1, 6)` inicializada con ceros para almacenar el resultado de la regla de Simpson aplicada a todo el intervalo. Luego se asigna el resultado de la función `srule` a `SRvec`.
   - Se inicializa `m` con el valor 1 para llevar el conteo de las iteraciones.
   - Se establece el estado en `ite`.
2. **Bucle principal**:
   - Se inicia un bucle `while` que se ejecutará mientras `state` sea igual a `ite`.
3. **Actualización del tamaño del intervalo**:
   - Se actualiza el valor de `n` con el valor actual de `m`, que representa el número de intervalos en la iteración actual.
4. **Bucle de iteración inverso**:

- Se inicia un bucle `for` que itera desde `n-1` hasta 0 en orden inverso. Esto permite realizar el proceso de adaptación en cada intervalo de manera descendente.

5. **Adaptación del intervalo**:
   - Se obtiene el vector `SR0vec` correspondiente al intervalo actual desde la matriz `SRmat`.
   - Se extraen los valores de error `err` y tolerancia `tol` del vector `SR0vec`.
   - Si la tolerancia es menor o igual al error, se realiza la adaptación del intervalo.
   - Se crean los vectores `SR1vec` y `SR2vec` para aplicar la regla de Simpson en los subintervalos divididos por `c`, el punto medio del intervalo actual.
   - Se calcula el nuevo error `err` y se actualiza.
   - Si el nuevo error es menor que la tolerancia, se actualizan los valores correspondientes en `SRmat`. En caso contrario, se insertan los nuevos vectores `SR1vec` y `SR2vec` en `SRmat` y se actualiza `m` y `state` para seguir con más iteraciones.
6. **Cálculo del resultado y errores:**
   - Se calcula el resultado total `quad` sumando los valores de la columna 3 de la matriz `SRmat`.
   - Se calcula el error total `err` sumando los valores de la columna 4 de la matriz `SRmat`.
   - Se recorta la matriz `SRmat` para eliminar las filas no utilizadas, dejando solo las filas con resultados válidos.
7. **Retorno de resultados**:
   - La parte final del proceso construye todos los resultados que hemos obtenido como la longitud del intervalo, la matriz de pasos y el resultado y los devuelve en formato tabla para que así sea más visual este apartado en la aplicación didáctica.`pasos`.

En resumen, la función **simpsonAdaptativo** implementa la regla de Simpson adaptativa para aproximar la integral de una función en un intervalo dado. Utiliza iteraciones y adaptación de los subintervalos para obtener una aproximación más precisa, dividiendo los intervalos en subintervalos más

pequeños hasta que se cumpla una tolerancia dada. El resultado final es la suma de las aproximaciones de los subintervalos y se devuelve junto con los pasos intermedios en una tabla en formato tabla.

Capítulo 4

Resultados y conclusiones

4.1. Funcionamiento de la aplicación

La aplicación didáctica es una herramienta diseñada para ayudar a comprender y visualizar los conceptos y métodos de las asignaturas de Matemáticas. El objetivo principal de esta parte de la aplicación es brindar una experiencia interactiva y educativa que permita a los estudiantes explorar y experimentar con diferentes técnicas de integración numérica.

La aplicación proporciona una interfaz intuitiva donde los usuarios pueden ingresar una función matemática y especificar el intervalo de integración. Además, la aplicación ofrece la posibilidad de ajustar el número de subintervalos utilizados en la aproximación para observar cómo afecta la precisión al resultado.

La aplicación muestra tanto el resultado final de la integral aproximada como los pasos intermedios del cálculo. Estos pasos incluyen información sobre cada subintervalo, como el número de iteración, su punto medio, el valor de la función evaluada en ese punto medio y la suma acumulativa hasta el paso actual.

Estos pasos buscan ayudar al alumno en sus cálculos en caso de que quisiera realizar las operaciones a mano con el objetivo de aprender las distintas técnicas. En caso de que el número de subintervalos sea demasiado alto, sólo se devolverán los nueve primeros pasos y el último a modo de ejemplo, ya que se entiende que los alumnos no van a realizar cálculos para un número elevado de subintervalos.

4.1.1. Propuestas de aplicaciones en el aula

La aplicación se presenta como una solución tanto para los estudiantes como para los profesores de asignaturas de la rama Cálculo de la Escuela Técnica Superior de Ingenieros Informáticos de la Universidad Politécnica de Madrid. Para los estudiantes, la aplicación brinda la oportunidad de explorar y experimentar con diferentes técnicas de integración numérica de forma interactiva, lo que puede ayudar a reforzar su comprensión y a aplicar los conocimientos teóricos en un entorno práctico.

Además, la aplicación ofrece una visualización clara de los pasos intermedios del cálculo, lo que permite a los estudiantes comprender mejor cómo se divide la integral en subintervalos y cómo se aproxima el área bajo la curva mediante la suma de áreas de polígonos más sencillos. Esto fomenta un aprendizaje más activo y participativo.

Por otro lado, la aplicación también puede resultar útil para los profesores al ahorrarles tiempo en la corrección de resultados y en el cálculo preciso de integrales numéricas cuando se utilizan un número elevado de subintervalos. Asimismo, la posibilidad de trabajar con herramientas computacionales en clase y en casa puede resultar más atractiva para los alumnos, especialmente en asignaturas teóricas, lo que puede motivarlos a explorar y utilizar diferentes recursos tecnológicos en su proceso de aprendizaje.

4.2. Posibles mejoras

Se proponen las siguientes mejoras para la aplicación didáctica de integración numérica:

1. **Visualización gráfica de la función y los subintervalos**: para fomentar la claridad y una comprensión más visual, se puede implementar una función en la aplicación que muestre el dibujo de la función original junto con los subintervalos utilizados en el cálculo de la integral. Esto permitirá a los estudiantes visualizar de manera intuitiva cómo se aproxima el área bajo la curva mediante la suma de áreas de rectángulos, facilitando una comprensión más profunda del proceso.
2. **Extensiones de la aplicación**: para enriquecer la experiencia educativa, se pueden agregar extensiones a la aplicación que brinden más opciones y herramientas a los usuarios. Por ejemplo, se podría proporcionar una sección de ejercicios interactivos que permita a los estudiantes practicar y aplicar los conceptos aprendidos.

3. **Presentación de información de manera más ilustrativa**: para hacer que la información sea más atractiva y fácil de entender, se pueden utilizar elementos visuales como gráficos, diagramas o animaciones interactivas para presentar los pasos intermedios del cálculo. Esto ayudará a los estudiantes a visualizar de manera más clara cómo se va acumulando la suma de áreas de rectángulos y cómo se llega al resultado final de la integral aproximada. Además, se pueden proporcionar explicaciones detalladas y ejemplos concretos para cada paso, lo que facilitará la comprensión y el seguimiento del proceso.

Estas mejoras permitirán que la aplicación didáctica de integración numérica sea más completa, interactiva y visualmente atractiva, brindando a los estudiantes una experiencia educativa enriquecedora y facilitando su comprensión y aplicación de los conceptos y técnicas de integración numérica.

Capítulo 5

Análisis de impacto

La Agenda 2030, adoptada por las Naciones Unidas en 2015, es un plan global para erradicar la pobreza, proteger el planeta y promover la paz y la prosperidad. Consta de 17 Objetivos de Desarrollo Sostenible (ODS) interconectados que abordan desafíos como la pobreza, la desigualdad y el cambio climático (véase [7]). Hay cuatro ODS principales que guardan una estrecha relación con este Trabajo de Fin de Grado (véase [8, 9, 10, 11]), son los siguientes:

- **Educación de calidad (ODS 4)**: el proyecto de creación de una aplicación didáctica busca mejorar el aprendizaje de las asignaturas de Matemáticas, brindando una herramienta de apoyo para los estudiantes. Contribuye a garantizar una educación inclusiva, equitativa y de calidad, promoviendo oportunidades de aprendizaje para todos.
- **Trabajo decente y crecimiento económico (ODS 8)**: la creación de una aplicación didáctica que mejora el aprendizaje de las Matemáticas, prepara a los estudiantes para el mercado laboral y fomenta el crecimiento económico sostenible.
- **Industria, innovación e infraestructura (ODS 9)**: el desarrollo de una aplicación especializada en el campo de las Matemáticas y su implementación mediante algoritmos demuestra el uso de tecnologías innovadoras. Contribuye a fomentar la infraestructura científica y tecnológica, promoviendo la investigación y el desarrollo tecnológico.
- **Producción y consumo responsables (ODS 12)**: al utilizar un software especializado para la resolución de ejercicios de Matemáticas, el proyecto promueve un enfoque responsable hacia el uso de recursos

y la reducción del consumo de papel. Contribuye a fomentar prácticas sostenibles en el ámbito educativo.

Bibliografía

[1] A. Doubova and F. Guillén-González, *Un Curso de Cálculo Numérico: Interpolación, Aproximación, Integración y Resolución de Ecuaciones Diferenciales*. Secretariado de Publicaciones, Universidad de Sevilla, 2007.

[2] J. H. Mathews and K. D. Fink, *Métodos Numéricos con MATLAB*. Prentice-Hall, 2000.

[3] R. L. Burden, D. J. Faires, and A. M. Burden, *Análisis Numérico*. Cengage Learning Editorial, 2017.

[4] "Numpy." [Online]. Disponible: https://numpy.org/doc/stable/

[5] "Sympy." [Online]. Disponible: https://www.sympy.org/en/index.html

[6] "Cálculo simbólico con sympy." [Online]. Disponible: http://research.iac.es/sieinvens/python-course/sympy.html

[7] "Objetivos de desarrollo sostenible (ODS)." [Online]. Disponible: https://www.mdsocialesa2030.gob.es/agenda2030/index.htm

[8] "Objetivo 4: Garantizar una educación inclusiva, equitativa y de calidad y promover oportunidades de aprendizaje durante toda la vida para todos." [Online]. Disponible: https://www.un.org/sustainabledevelopment/es/education/

[9] "Objetivo 8: Promover el crecimiento económico inclusivo y sostenible, el empleo y el trabajo decente para todos." [Online]. Disponible: https://www.un.org/sustainabledevelopment/es/economic-growth/

[10] "Objetivo 9: Construir infraestructuras resilientes, promover la industrialización sostenible y fomentar la innovación." [Online]. Disponible: https://www.un.org/sustainabledevelopment/es/infrastructure/

[11] “Objetivo 12: Garantizar modalidades de consumo y producción sostenibles.” [Online]. Disponible: https://www.un.org/sustainabledevelopment/es/sustainable-consumption-production/

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Thu Jun 08 15:14:11 CEST 2023
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)