EST. 2021 EMC
EDITORIAL MAR CARIBE

Métodos numéricos aplicados con software:

Sintaxis en GNU Octave y Python

2025

RAPHAEL SANTIAGO MENDOZA DELGADO
JORGE LUIS ROJAS ORBEGOSO
JORGE LUIS ILQUIMICHE MELLY
ROBERTO JAVIER VARGAS QUINTANA
JOSÉ RICARDO RASILLA ROVEGNO
CESAR AUGUSTO ONTIVEROS BOHORQUEZ
JONHY SATURNINO GARAY SANTISTEBAN

ISBN: 978-9915-698-16-8
9 789915 698168

Métodos numéricos aplicados con software: Sintaxis en GNU Octave y Python

Raphael Santiago Mendoza Delgado, Jorge Luis Rojas Orbegoso, Jorge Luis Ilquimiche Melly, Roberto Javier Vargas Quintana, José Ricardo Rasilla Rovegno, Cesar Augusto Ontiveros Bohorquez, Jonhy Saturnino Garay Santisteban

Primera edición: Junio, 2025

Editado por:

Editorial Mar Caribe

www.editorialmarcaribe.es

Av. General Flores 547, Colonia, Colonia-Uruguay.

Diseño de portada: Yelitza Sánchez Cáceres

Libro electrónico disponible en:

https://editorialmarcaribe.es/ark:/10951/isbn.9789915698168

Formato: electrónico

ISBN: 978-9915-698-16-8

ARK: ark:/10951/isbn.9789915698168

Editorial Mar Caribe

Métodos numéricos aplicados con software: Sintaxis en GNU Octave y Python

Colonia, Uruguay

2025

Sobre los autores y la publicación

Raphael Santiago Mendoza Delgado

https://orcid.org/0009-0003-3679-0809

Universidad Nacional del Callao, Perú

Jorge Luis Rojas Orbegoso

https://orcid.org/0000-0002-5688-4963

Universidad Nacional del Callao, Perú

Jorge Luis Ilquimiche Melly

https://orcid.org/0000-0001-5974-1979

Universidad César Vallejo, Perú

Roberto Javier Vargas Quintana

https://orcid.org/0000-0001-9790-0168

Universidad Nacional San Luis Gonzaga, Perú

José Ricardo Rasilla Rovegno

https://orcid.org/0009-0006-4747-1864

Universidad Nacional del Callao, Perú

Cesar Augusto Ontiveros Bohorquez

https://orcid.org/0009-0006-2287-5256

Universidad Nacional José Faustino Sánchez Carrión, Perú

Jonhy Saturnino Garay Santisteban

https://orcid.org/0000-0001-6329-4438

Universidad Nacional Santiago Antúnez de Mayolo, Perú

Resultado de la investigación del libro:

Publicación original e inédita, cuyo contenido es el resultado de un proceso de investigación realizado antes de su publicación, ha sido doble ciego de revisión externa por pares, el libro ha sido seleccionado por su calidad científica y porque contribuye significativamente al área del conocimiento e ilustra una investigación completamente desarrollada y completada. Además, la publicación ha pasado por un proceso editorial que garantiza su estandarización bibliográfica y usabilidad.

Sugerencia de citación: Mendoza, R.S., Rojas, J.L., Ilquimiche, J.L., Vargas, R.J., Rasilla, J.R., Ontiveros, C.A., y Garay, J.S. (2025). *Métodos numéricos aplicados con software: Sintaxis en GNU Octave y Python.* Colonia del Sacramento: Editorial Mar Caribe. https://editorialmarcaribe.es/ark:/10951/isbn.9789915698168

Índice

Introducción

Las técnicas numéricas constituyen una colección de técnicas matemáticas que convierten problemas en problemas matemáticos para resolverlos mediante métodos aproximados. Mientras que las soluciones analíticas monolíticas dan un valor exacto, los métodos numéricos permiten abordar problemas muy complejos que son imposibles de resolver directamente, especialmente cuando incluyen modelos numéricos con ecuaciones diferenciales, integrales y sistemas con ecuaciones lineales y no lineales.

La importancia de los métodos numéricos en la computación radica en que permiten soluciones prácticas cuando los modelos matemáticos son complejos por naturaleza. En la sociedad tecnológica y científica actual, donde las nuevas tecnologías emergen muy rápidamente, ingenieros, científicos y analistas de datos utilizan estas técnicas para realizar simulaciones, optimizaciones y análisis de un aspecto central en el desarrollo de nuevas tecnologías y la comprensión de eventos naturales.

El uso de métodos numéricos se ha vuelto común a medida que el poder de cómputo y el software dedicado han aumentado. Profesionales y estudiantes han tenido la libertad de explotar estas herramientas para resolver problemas en áreas que van desde la física y la ingeniería hasta la economía y la biología. Los enfoques computacionales no solo ayudan a resolver problemas difíciles, sino que también algunos de ellos están abriendo un nuevo punto de vista para explorar y visualizar datos que antes no se consideraban posibles.

Este libro introduce los métodos numéricos en el contexto de resolver problemas complejos y acertijos diseñados para ilustrar la riqueza del mundo numérico. En el curso de estudiar las herramientas GNU Octave y Python, también obtendremos una visión sobre la implementación de estos métodos en la práctica, así como sobre las preguntas de qué herramienta es preferible para qué tarea. En este sentido, el objetivo de investigación es ampliar los objetivos del análisis numérico con Octave y Python, integrando tanto los

fines académicos y prácticos del análisis numérico como las ventajas y limitaciones concretas del software y su ecosistema científico.

GNU Octave es un lenguaje de alto nivel, destinado principalmente a cálculos numéricos y análisis de datos. Su sintaxis es similar a la de MATLAB, lo que facilita relativamente el cambio para los usuarios de MATLAB. Aquí discutimos algunas técnicas numéricas elementales que se pueden realizar en GNU Octave, incluyendo la interpolación y la resolución de ecuaciones diferenciales. Al iniciar Octave, los usuarios se encuentran con una interfaz de línea de comandos interactiva, que les permite ejecutar fragmentos de código (en forma de scripts y funciones) así como comandos e importar/exportar datos. Octave también se puede usar para escribir y ver gráficos o visualizar datos, ambos importantes en el análisis numérico.

Python también ha surgido como un lenguaje de programación popular para la ciencia de datos y el cálculo numérico debido a su sintaxis limpia y la gran cantidad de bibliotecas especializadas. Durante el desarrollo de este libro, vamos a estudiar algunas bibliotecas influyentes capaces de trabajar con métodos numéricos en Python con lecciones sobre lo que se puede hacer con ellas.

Entre GNU Octave y Python será una cuestión de cuáles son los requisitos del usuario y en qué escenario (campo) se van a aplicar los métodos numéricos. GNU Octave es la mejor opción de programación rápida, pero si se necesita flexibilidad y capacidad de integración, es recomendable usar Python. Eventualmente, ambas herramientas proporcionan un buen complemento para la construcción de métodos numéricos, y la preferencia también puede depender de las preferencias o familiaridad con el lenguaje de programación.

Ambos softwares tienen sus propias ventajas y desventajas. GNU Octave es excelente para cualquiera que conozca MATLAB y necesite un entorno de software numérico rápido. Python también es la mejor opción para entornos de desarrollo complejos debido a su flexibilidad e integrabilidad. Basado en la experiencia del usuario, las necesidades del proyecto y el apoyo de la comunidad, la selección del software apropiado será considerada por los autores. Los usuarios podrán leer tanto las características como sus

aplicaciones en Octave y Python para que puedan tomar una decisión informada y maximizar su trabajo en el área de métodos numéricos.

Capítulo I

Métodos Numéricos con GNU Octave y Python: Una Guía Práctica para Interpolación, Integración y Resolución de Ecuaciones

Los métodos numéricos son técnicas matemáticas que permiten resolver problemas que, en su forma analítica, pueden ser complejos o incluso imposibles de abordar. Estos métodos son fundamentales en el ámbito de la ingeniería, la física, la economía y muchas otras disciplinas que requieren la resolución de ecuaciones, el análisis de datos o la simulación de fenómenos.

La importancia de los métodos numéricos radica en su capacidad para proporcionar soluciones aproximadas a problemas matemáticos que no pueden ser resueltos de forma exacta. Así, en el caso de ecuaciones diferenciales que modelan fenómenos físicos, a menudo es imposible encontrar una solución cerrada; en cambio, los métodos numéricos nos permiten aproximar estas soluciones con un nivel de precisión que puede ser ajustado según las necesidades del problema en cuestión.

Además, el avance de la tecnología informática ha facilitado la implementación de estos métodos en software específico, como GNU Octave y Python. Estos programas permiten a los investigadores y profesionales aplicar algoritmos numéricos de manera eficiente, optimizando el tiempo de cálculo y aumentando la accesibilidad de herramientas avanzadas para el análisis y la simulación. Los métodos numéricos son una herramienta esencial en el arsenal de cualquier científico o ingeniero, su capacidad para transformar problemas complejos en soluciones prácticas ha revolucionado la forma en que abordamos el análisis matemático, y su integración con potentes lenguajes de programación amplía aún más sus aplicaciones.

1.1 Métodos de interpolación

La interpolación es una técnica fundamental en los métodos numéricos que se utiliza para estimar valores intermedios de una función a partir de un conjunto de puntos discretos. Su importancia radica en la capacidad de

modelar datos experimentales, realizar predicciones y simplificar el análisis de funciones complejas. La interpolación consiste en encontrar una función que pase a través de un conjunto de puntos conocidos, denominados nodos. Dicha función se utiliza para estimar valores en puntos donde no se dispone de datos. Existen múltiples métodos de interpolación, siendo la interpolación polinómica uno de los más comunes (Zota, 2015). Este método utiliza polinomios para aproximar la función, lo que permite obtener valores intermedios con un grado de precisión aceptable, especialmente cuando los puntos de datos son cercanos entre sí.

Las aplicaciones de la interpolación son diversas y se extienden a campos como la ingeniería, la economía, la ciencia de datos y la investigación científica. En efecto, en la ingeniería, la interpolación se utiliza para calcular propiedades de materiales a partir de datos experimentales, mientras que en la ciencia de datos se aplica para suavizar curvas y mejorar la visualización de tendencias en datos.

GNU Octave es un entorno de programación de alto nivel especialmente diseñado para el cálculo numérico, que ofrece una sintaxis similar a MATLAB. Para implementar la interpolación polinómica en GNU Octave, se puede utilizar la función polyfit para ajustar un polinomio a un conjunto de datos y polyval para evaluar el polinomio en puntos deseados (Borrell, 2008). Posteriormente, se presenta un ejemplo práctico de interpolación polinómica en Octave:

```
octave

% Datos de ejemplo

x = [1, 2, 3, 4, 5];

y = [2.2, 3.8, 5.5, 7.1, 8.3];


% Ajuste de un polinomio de grado 2

p = polyfit(x, y, 2);


% Evaluación del polinomio en nuevos puntos
```

```
x_new = linspace(1, 5, 100);
y_new = polyval(p, x_new);

% Gráfica de los resultados
plot(x, y, 'o', x_new, y_new, '-');
title('Interpolación Polinómica en GNU Octave');
xlabel('x');
ylabel('y');
legend('Datos originales', 'Polinomio ajustado');
grid on;
```

En este código, se definen los puntos de datos, se ajusta un polinomio de grado 2 a esos datos y se evalúa el polinomio en un rango más amplio de valores de x para visualizar la interpolación. Python, junto con la biblioteca NumPy, ofrece herramientas poderosas para realizar interpolación polinómica. La función numpy.polyfit permite ajustar un polinomio a un conjunto de datos, en tanto que numpy.polyval se utiliza para evaluar el polinomio en puntos específicos. A continuación, se exhibe un ejemplo de cómo realizar la interpolación polinómica en Python:

```
python
import numpy as np
import matplotlib.pyplot as plt

 Datos de ejemplo
x = np.array([1, 2, 3, 4, 5])
y = np.array([2.2, 3.8, 5.5, 7.1, 8.3])
```

```
Ajuste de un polinomio de grado 2
p = np.polyfit(x, y, 2)

Evaluación del polinomio en nuevos puntos
x_new = np.linspace(1, 5, 100)
y_new = np.polyval(p, x_new)

Gráfica de los resultados
plt.plot(x, y, 'o', label='Datos originales')
plt.plot(x_new, y_new, '-', label='Polinomio ajustado')
plt.title('Interpolación Polinómica en Python')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid()
plt.show()
```

En este ejemplo, se utilizan las mismas coordenadas de datos que en el ejemplo de Octave. Se ajusta un polinomio de grado 2 y se visualiza el resultado utilizando Matplotlib, una biblioteca popular para la creación de gráficos en Python. A través de estos ejemplos en GNU Octave y Python, se demuestra la versatilidad y la facilidad de implementación de los métodos de interpolación polinómica en diferentes entornos de programación, lo cual es esencial para el análisis y la modelización de datos en diversas disciplinas.

La integración numérica se utiliza para aproximar el valor de integrales definidas, especialmente cuando las funciones a integrar son complejas o no

tienen una antiderivada explícita. La integración es esencial en diversas aplicaciones, desde la física y la ingeniería hasta la economía, ya que permite calcular áreas bajo curvas, volúmenes y otros valores que son difíciles de obtener de manera analítica. A través de la integración numérica, podemos obtener estimaciones precisas de estas áreas y volúmenes utilizando técnicas computacionales.

La integración numérica se basa en dividir el área bajo una curva en segmentos más pequeños que son más fáciles de calcular, existen diversas técnicas para llevar a cabo esta aproximación, siendo las más comunes la regla del trapecio y la regla de Simpson. Ambas técnicas utilizan polinomios para aproximar la función a integrar, permitiendo así que el cálculo del área se realice de manera más eficiente en comparación con los métodos analíticos.

La regla del trapecio consiste en aproximar la función mediante líneas rectas, formando trapecios en lugar de utilizar la curva completa: por otro lado, la regla de Simpson utiliza parábolas para realizar la aproximación, lo que proporciona una estimación más precisa, especialmente cuando la función es suave y continua (Nakamura, 1992). Ambas técnicas son ampliamente utilizadas en software como GNU Octave y Python, donde se pueden implementar de manera sencilla y eficiente. En GNU Octave, la regla del trapecio se puede implementar de manera sencilla utilizando funciones integradas. A continuación, se presenta un ejemplo básico de cómo aplicar este método:

```
octave

% Definimos la función a integrar

f = @(x) x.^2;  % Ejemplo: f(x) = x^2

a = 0;          % Límite inferior

b = 1;          % Límite superior

n = 100;        % Número de subintervalos


% Cálculo de la integral usando la regla del trapecio
```

```
x = linspace(a, b, n+1);  % Puntos de evaluación
y = f(x);                 % Evaluamos la función en esos puntos
integral_trapecio = trapz(x, y);  % Aplicamos la regla del trapecio

% Mostramos el resultado
disp(['La integral aproximada es: ', num2str(integral_trapecio)]);
```

En este código, definimos la función \(f(x) = x^2 \) y utilizamos la función trapz de Octave para calcular la integral aproximada en el intervalo \([0, 1]\). Al aumentar el número de subintervalos n, se mejora la precisión del resultado. En Python, la biblioteca SciPy ofrece una función conveniente para aplicar la regla de Simpson, lo que facilita la integración numérica. En pos se muestra un ejemplo de cómo realizar este cálculo:

```
python
import numpy as np
from scipy.integrate import simpson

 Definimos la función a integrar
def f(x):
    return x  2   Ejemplo: f(x) = x^2

a = 0            Límite inferior
b = 1            Límite superior
n = 100          Número de subintervalos, debe ser par
```

```
Generamos los puntos de evaluación
x = np.linspace(a, b, n + 1)
y = f(x)          Evaluamos la función en esos puntos

Cálculo de la integral usando la regla de Simpson
integral_simpson = simpson(y, x)

Mostramos el resultado
print(f'La integral aproximada es: {integral_simpson}')
```

En este ejemplo, definimos la misma función \(f(x) = x^2 \) y utilizamos la función simpson de SciPy para calcular la integral en el intervalo \([0, 1]\). Al igual que con la regla del trapecio, se puede mejorar la precisión variando el número de subintervalos n, que debe ser par para aplicar correctamente la regla de Simpson. Los métodos de integración numérica son herramientas poderosas en el análisis matemático y la ingeniería. Tanto GNU Octave como Python ofrecen funciones eficientes para implementar estas técnicas, lo que permite a los usuarios obtener resultados precisos y realizar análisis complejos de manera efectiva.

La resolución de ecuaciones es una de las tareas fundamentales en el ámbito de las matemáticas aplicadas y la ingeniería. A menudo, se enfrentan a problemas que requieren encontrar las raíces de funciones, es decir, los valores de \(x \) que satisfacen la ecuación \(f(x) = 0 \). Existen varios tipos de ecuaciones, que pueden clasificarse en ecuaciones algebraicas, transcendentes y diferenciales, entre otras. La selección del método más adecuado para resolver una ecuación depende de la naturaleza de la función, la cantidad de soluciones esperadas y la precisión requerida.

Las ecuaciones algebraicas son aquellas que pueden expresarse en términos de polinomios, mientras que las ecuaciones transcendentes incluyen funciones como exponenciales, logaritmos y trigonométricas. Los métodos de

resolución de ecuaciones pueden clasificarse en dos categorías principales: métodos analíticos y métodos numéricos. Los métodos analíticos, cuando son posibles, proporcionan soluciones exactas, pero no siempre se pueden aplicar a todas las funciones. Por otro lado, los métodos numéricos ofrecen aproximaciones a estas soluciones y son especialmente útiles en casos donde los métodos analíticos fallan o son impracticables.

Algunos de los métodos numéricos más comunes incluyen el método de Newton-Raphson, que es un enfoque iterativo que utiliza derivadas para encontrar raíces, y el método de bisección, que es un método más simple que divide el intervalo en el que se busca la raíz. Ambos métodos tienen sus aplicaciones y ventajas según el contexto en que se utilicen. El método de Newton-Raphson es especialmente eficaz para funciones que son continuamente diferenciables, se basa en la idea de utilizar la tangente a la curva de la función en un punto inicial para aproximar la raíz. La fórmula iterativa es:

\[
x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}
\]

donde \(f \) es la función y \(f' \) su derivada. En GNU Octave, la implementación de este método puede hacerse de la siguiente manera:

```
octave
function x = newton_raphson(f, df, x0, tol, max_iter)
  x = x0;
  for i = 1:max_iter
    x_new = x - f(x) / df(x);
    if abs(x_new - x) < tol
```

```
        break;
    end
    x = x_new;
  end
end
```

Aquí, f es la función cuya raíz buscamos, df es la derivada de la función, x0 es la estimación inicial, tol es la tolerancia para la convergencia y max_iter es el número máximo de iteraciones permitidas. El método de bisección es un enfoque más sencillo y robusto que garantiza la convergencia siempre que la función sea continua en el intervalo considerado y que se tenga un cambio de signo. Consiste en dividir el intervalo en dos y evaluar en cuál de los subintervalos se encuentra la raíz. En Python, utilizando la biblioteca NumPy, podemos implementar este método de la siguiente manera:

```
python
import numpy as np

def bisection(f, a, b, tol, max_iter):
    if f(a) * f(b) >= 0:
        raise ValueError("La función debe tener un cambio de signo en el intervalo [a, b].")

    for i in range(max_iter):
        c = (a + b) / 2
        if abs(f(c)) < tol or (b - a) / 2 < tol:
            return c
```

```
        if f(c) * f(a) < 0:
            b = c
        else:
            a = c
    return (a + b) / 2
```

En este fragmento de código, f es la función a evaluar, a y b son los límites del intervalo, tol es la tolerancia y max_iter es el número máximo de iteraciones permitidas. Este método es particularmente útil en aplicaciones donde se requiere un alto grado de certeza en la solución. Los métodos de resolución de ecuaciones son herramientas esenciales en el análisis numérico, y su implementación en software como GNU Octave y Python permite a los usuarios resolver problemas complejos de manera eficiente.

Además, el desarrollo de software y bibliotecas más robustas en lenguajes como Python, junto con la accesibilidad de plataformas de código abierto como GNU Octave, permitirá a una mayor cantidad de investigadores y profesionales aplicar métodos numéricos en sus trabajos. Esto no solo democratiza el acceso a estas herramientas, sino que también fomenta la colaboración interdisciplinaria, donde expertos de diferentes campos pueden contribuir al avance de las técnicas numéricas. Los métodos numéricos seguirán siendo un componente esencial en la investigación y el desarrollo en la era digital, la combinación de teoría sólida y aplicaciones prácticas en entornos de programación facilitará un avance continuo en la eficacia de los métodos numéricos.

1.2 Comparativa de Métodos Iterativos de Jacobi y Gauss-Siedel: Implementación en GNU Octave y Python

Los métodos iterativos son una clase de algoritmos ampliamente utilizados en la resolución de problemas matemáticos, especialmente en el contexto de sistemas lineales. A diferencia de los métodos directos, que buscan una solución exacta en un número finito de pasos, los métodos iterativos abordan el problema mediante un proceso repetitivo, donde cada

iteración busca acercar la solución a un valor esperado (Ferreira et al., 2025). Este enfoque es particularmente útil en situaciones donde el tamaño del sistema es grande o la matriz del sistema presenta ciertas propiedades que dificultan su resolución directa.

Un método iterativo se basa en la idea de iniciar con una estimación inicial de la solución y luego refinar esa estimación a través de una serie de pasos sucesivos. La solución se aproxima cada vez más con cada iteración, hasta que se alcanza un criterio de convergencia predefinido, como un error aceptable o un número máximo de iteraciones. Este proceso permite que los métodos iterativos sean flexibles y adaptables a diferentes tipos de problemas y condiciones.

La resolución de sistemas lineales es un problema fundamental en matemáticas aplicadas, ingeniería y ciencias computacionales, en muchos casos, estos sistemas están representados por matrices grandes y dispersas, donde los métodos directos pueden resultar ineficientes o impracticables debido al alto costo computacional (Solís et al., 2021). Aquí es donde los métodos iterativos, como Jacobi y Gauss-Siedel, juegan un papel trascendental. Proporcionan una alternativa efectiva y a menudo más rápida para encontrar soluciones aproximadas, especialmente en sistemas donde las matrices son difíciles de manejar.

El método de Jacobi es un algoritmo iterativo utilizado para resolver sistemas de ecuaciones lineales de la forma $Ax = b$, donde A es una matriz cuadrada, x es el vector de incógnitas y b es el vector de términos independientes. La esencia del método radica en la descomposición de la matriz A en su parte diagonal y el resto de los elementos. La idea básica es que, en cada iteración, se calcula el nuevo valor de cada incógnita utilizando los valores más recientes disponibles de las demás incógnitas. Matemáticamente, el método se puede expresar como:

$$
x^{(k+1)}_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x^{(k)}_j \right)
$$

donde \(x^{(k)}\) es el vector de solución en la \(k\)-ésima iteración. Este método es especialmente útil en sistemas donde la matriz \(A\) es diagonal dominante, lo que facilita la convergencia del algoritmo. La implementación del método de Jacobi en GNU Octave es relativamente sencilla. En seguida se presenta un ejemplo básico de cómo se puede programar:

```
octave
function [x]=Jacobi()
 A=[2 5 -8; 6 2 -3; 2 -9 5]%matriz de sistema, debe ser de diagonal dominante
 b=[1; 5; 9]%lado derecho
 x0=[0; 0; 0]%vector de valores iniciales
 tol=10^(-10) %tolerancia
 N=10%numero maximo de iteraciones
 format rat
 n=size(A);
 for k=1:N
  for i=1:n
   x(i,1)=(b(i,1)-A(i,:)*x0+A(i,i)*x0(i,1))/A(i,i);
  end
  printf("Iteracion %d\n",k)
  x
  err=norm(A*x-b);
  printf("El error es %f\n",err)
  if err<tol
   printf("La solucion due aproximada en %d iteraciones\n",k)
```

```
    break %rompe el for de k
  else
    x0=x;
  end
end
if k>N
  printf("El metodo fracaso despues de %d iteraciones \n",k)
endif
x2=A\b; %solucion exacta calculado por eliminacion gaussiana
err2=norm(x-x2);
printf("La distancia entre la sol aproximada y la exacta es %f\n",err2)
format short
endfunction
```

En este código, A es la matriz de coeficientes, b es el vector de términos independientes, x0 es una estimación inicial del vector de soluciones, tol es la tolerancia para la convergencia y max_iter es el número máximo de iteraciones permitidas. La implementación del método de Jacobi en Python puede realizarse utilizando bibliotecas como NumPy para facilitar las operaciones matriciales. Posteriormente se muestra un ejemplo de cómo se puede implementar:

```
python
import numpy as np

def jacobi(A, b, tolerancia=1.0e-8, max_iteraciones=100):
    """
```

```
    Resuelve un sistema de ecuaciones lineales Ax = b usando el método de Jacobi.

    Args:
        A: Matriz de coeficientes (numpy array).
        b: Vector de términos independientes (numpy array).
        tolerancia: Error máximo permitido para la convergencia.
        max_iteraciones: Número máximo de iteraciones permitidas.

    Returns:
        x: Vector solución (numpy array) o None si no converge.
        iteraciones: Número de iteraciones realizadas.
    """
    n = len(b)
    x = np.zeros(n)  # Inicialización con ceros
    x_nuevo = np.zeros(n)
    iteraciones = 0

    while iteraciones < max_iteraciones:
        for i in range(n):
            suma = 0
            for j in range(n):
                if i != j:
                    suma += A[i, j] * x[j]
            x_nuevo[i] = (b[i] - suma) / A[i, i]
```

```
        # Criterio de convergencia
        if np.linalg.norm(x_nuevo - x) < tolerancia:
            return x_nuevo, iteraciones

        x = np.copy(x_nuevo)  # Actualizar x para la próxima iteración
        iteraciones += 1

    print("El método no converge dentro del número máximo de iteraciones.")
    return None, max_iteraciones

# Ejemplo de uso
if __name__ == '__main__':
    # Definir la matriz A y el vector b
    A = np.array([[10, 2, 1],
                  [1, 5, 1],
                  [2, 3, 10]], dtype=float)
    b = np.array([7, -8, 6], dtype=float)

    # Resolver el sistema
    solucion, iteraciones = jacobi(A, b)

    if solucion is not None:
        print("Solución encontrada:")
        print(solucion)
```

```
print(f"Número de iteraciones: {iteraciones}")


# Verificar la solución (opcional)
error = np.linalg.norm(np.dot(A, solucion) - b)
print(f"Error de la solución: {error}")
```

En este código, se utiliza np.dot para realizar el producto escalar y np.linalg.norm para calcular la norma infinita del vector de diferencias, permitiendo así evaluar la convergencia del método de manera efectiva. El método de Jacobi, a pesar de ser simple, proporciona una base sólida para entender los métodos iterativos y su implementación en diferentes lenguajes de programación (Ferreira et al., 2025).

El método de Gauss-Siedel es un algoritmo iterativo utilizado para resolver sistemas de ecuaciones lineales. Se basa en la idea de que, en cada iteración, se actualiza el valor de cada variable utilizando los valores más recientes disponibles. A diferencia del método de Jacobi, que utiliza los valores de la iteración anterior para todas las variables, el método de Gauss-Siedel permite que cada variable se actualice inmediatamente después de ser calculada.

Esta propiedad puede llevar a una convergencia más rápida en muchos casos, especialmente cuando se aplican a matrices que cumplen con ciertas condiciones, como la diagonalmente dominante. La formulación del método implica reordenar el sistema de ecuaciones de tal manera que se aísle una de las variables. Esto se hace de la siguiente manera: para un sistema de ecuaciones lineales $Ax = b$, se puede expresar la i-ésima variable como:

$$ x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) $$

donde \(x_j^{(k)} \) son los valores de la j-ésima variable en la k-ésima iteración. Este proceso se repite hasta que se alcanza un criterio de convergencia predefinido, como un número fijo de iteraciones o una tolerancia en el error. La implementación del método de Gauss-Siedel en GNU Octave es bastante directa y se puede realizar con unas pocas líneas de código. A continuación se presenta un ejemplo de cómo se podría implementar este método:

```
octave
function [x, iter] = gauss_seidel(A, b, tol, max_iter)
    n = length(b);
    x = zeros(n, 1);
    iter = 0;

    for iter = 1:max_iter
        x_old = x;
        for i = 1:n
            sum1 = A(i, 1:i-1)  x(1:i-1);
            sum2 = A(i, i+1:n)  x_old(i+1:n);
            x(i) = (b(i) - sum1 - sum2) / A(i, i);
        end
        if norm(x - x_old, inf) < tol
            break;
        end
    end
end
```

En este código, A es la matriz de coeficientes del sistema, b es el vector de términos independientes, tol es la tolerancia para detener la iteración y max_iter es el número máximo de iteraciones permitidas. La función devuelve el vector solución x y el número de iteraciones realizadas. La implementación del método de Gauss-Siedel en Python puede lograrse utilizando bibliotecas como NumPy, que facilita las operaciones matriciales, se exhibe un ejemplo de cómo implementar el método:

```
python
import numpy as np

def gauss_seidel(A, b, tol=1e-10, max_iter=100):
    n = len(b)
    x = np.zeros(n)

    for iter in range(max_iter):
        x_old = np.copy(x)
        for i in range(n):
            sum1 = np.dot(A[i, :i], x[:i])
            sum2 = np.dot(A[i, i+1:], x_old[i+1:])
            x[i] = (b[i] - sum1 - sum2) / A[i, i]
        if np.linalg.norm(x - x_old, np.inf) < tol:
            break

    return x, iter
```

En este código, se utiliza np.dot para realizar la multiplicación de matrices y np.linalg.norm para calcular la norma del vector de errores. Al igual que en el caso de GNU Octave, se proporciona la capacidad de especificar una tolerancia y un número máximo de iteraciones antes de que el algoritmo se detenga. El método de Gauss-Siedel es una herramienta potente y eficiente para resolver sistemas de ecuaciones lineales, especialmente en contextos donde se requiere una rápida convergencia y se dispone de matrices que cumplen ciertas condiciones. Su implementación en GNU Octave y Python es accesible y permite a los usuarios aplicar este método en diversas aplicaciones prácticas.

La eficiencia de los métodos de Jacobi y Gauss-Siedel puede ser evaluada en función de varios criterios, entre los que se incluyen el número de iteraciones necesarias para alcanzar una solución aceptable y el tiempo de cálculo requerido. En general, el método de Gauss-Siedel tiende a ser más eficiente que el método de Jacobi. Esto se debe a que, en cada iteración de Gauss-Siedel, se utilizan los valores más recientes de las variables, lo que puede acelerar la convergencia hacia la solución. Por otro lado, el método de Jacobi calcula todos los nuevos valores de las variables utilizando únicamente los resultados de la iteración anterior, lo que puede resultar en un mayor número de iteraciones para alcanzar la convergencia.

La convergencia de ambos métodos depende de las propiedades de la matriz del sistema lineal que se está resolviendo. Para que el método de Jacobi converja, es necesario que la matriz sea diagonalmente dominante o que sea simétrica definida positiva. El método de Gauss-Siedel de igual modo converge bajo condiciones similares, si bien es más robusto en la práctica. En muchos casos, el método de Gauss-Siedel puede converger incluso si la matriz no es estrictamente diagonalmente dominante, lo que puede ser una ventaja en situaciones donde se trabaja con matrices más complicadas (Gil, 2006).

En el ámbito de la resolución de sistemas lineales, tanto el método de Jacobi como el de Gauss-Siedel tienen aplicaciones prácticas en diversas disciplinas, como la ingeniería y la física. El método de Jacobi es particularmente útil en situaciones donde el paralelismo es una consideración importante, ya que permite que las iteraciones se realicen de manera independiente. Esto lo hace ideal para implementaciones en hardware

especializado, como matrices de procesamiento paralelo. Por otro lado, el método de Gauss-Siedel se utiliza comúnmente en aplicaciones donde la rapidez de convergencia es fundamental, como en la simulación de fenómenos físicos o en el análisis estructural.

En síntesis, la elección entre el método de Jacobi y el de Gauss-Siedel dependerá de las características específicas del sistema que se esté resolviendo y de las prioridades en términos de eficiencia y convergencia. Ambos métodos tienen sus ventajas y desventajas, y es importante considerar el contexto y las necesidades del problema a resolver al decidir cuál método utilizar.

El método de Jacobi se caracteriza por su simplicidad y facilidad de implementación, lo que lo convierte en una excelente opción para problemas donde la convergencia es garantizada. Sin embargo, su velocidad de convergencia puede ser inferior en comparación con el método de Gauss-Siedel, que, al utilizar los resultados más recientes de las iteraciones, suele converger más rápidamente. Esta diferencia en la eficiencia destaca la importancia de elegir el método adecuado según las características del sistema a resolver.

En cuanto a la implementación, hemos proporcionado ejemplos tanto en GNU Octave como en Python, dos lenguajes ampliamente utilizados en el ámbito de la ciencia de datos y la computación. Estas implementaciones permiten a los usuarios familiarizarse con la sintaxis y la lógica detrás de cada método, facilitando su aplicación en proyectos reales. Se están desarrollando nuevas técnicas y algoritmos que podrían mejorar aún más la eficiencia y la rapidez de convergencia de los métodos existentes. Por lo tanto, recomendamos a los lectores mantenerse informados sobre las últimas tendencias y avances en este campo.

Los métodos de Jacobi y Gauss-Siedel son herramientas valiosas en la resolución de sistemas lineales. Su comprensión y correcta aplicación pueden facilitar la resolución de problemas complejos en diversas disciplinas. Con un conocimiento sólido de estos métodos y sus implementaciones, los profesionales y estudiantes podrán abordar con mayor confianza los desafíos que se presenten en su trabajo.

1.3 Método de Runge-Kutta: Implementación y Comparativa en GNU Octave y Python

El método de Runge-Kutta es una de las técnicas más utilizadas para aproximar soluciones de ecuaciones diferenciales ordinarias (EDOs). Su desarrollo ha permitido a matemáticos e ingenieros abordar problemas complejos que, de otro modo, serían intratables mediante métodos analíticos tradicionales. El método de Runge-Kutta se refiere a una familia de métodos de integración numérica que buscan proporcionar una solución aproximada a EDOs de la forma $y' = f(t, y)$, donde y es la función desconocida y f es una función conocida que depende de t y y. La idea central detrás de estos métodos es calcular múltiples pendientes en un intervalo y utilizar estas pendientes para estimar el valor de la función en el siguiente paso. Esto permite una mayor precisión en comparación con métodos más simples, como el método de Euler, que solo utiliza la pendiente en un único punto.

El origen de los métodos de Runge-Kutta se remonta a principios del siglo XX, cuando los matemáticos Carl Runge y Wilhelm Kutta, por separado, desarrollaron técnicas para resolver EDOs. La primera versión del método, conocida como el método de Runge-Kutta de cuarto orden, fue formulada a principios de 1900 y se ha convertido en el más popular debido a su equilibrio entre precisión y complejidad computacional (Mata, 2016). Desde entonces, se han desarrollado diversas variantes del método, cada una optimizada para diferentes tipos de EDOs y requisitos computacionales.

El método de Runge-Kutta es fundamental en la resolución de EDOs por varias razones. En primer lugar, su capacidad para manejar problemas no lineales y sistemas de ecuaciones diferenciales lo convierte en una herramienta versátil en campos como la física, la ingeniería y la biología. Asimismo, su implementación es relativamente sencilla en lenguajes de programación populares, lo que permite a los investigadores y estudiantes utilizarlo sin una profunda comprensión de la teoría subyacente. Por último, la precisión y estabilidad de los métodos de Runge-Kutta los hacen adecuados para simular sistemas dinámicos complejos, lo que es esencial en la modelización de fenómenos del mundo real. GNU Octave es un software de programación de alto nivel, principalmente destinado a cálculos numéricos.

Su sintaxis es similar a la de MATLAB, lo que lo convierte en una herramienta accesible y potente para la implementación del método de Runge-Kutta. El método de Runge-Kutta de cuarto orden (RK4) es uno de los más utilizados debido a su equilibrio entre precisión y complejidad computacional. La implementación del RK4 en GNU Octave generalmente sigue la siguiente estructura:

```
octave
function y = runge_kutta_4(f, y0, t0, tf, h)
  % f: función que representa la ecuación diferencial
  % y0: valor inicial
  % t0: tiempo inicial
  % tf: tiempo final
  % h: tamaño del paso

  N = (tf - t0) / h; % Número de pasos
  y = zeros(1, N + 1); % Vector para almacenar los resultados
  y(1) = y0; % Asignar valor inicial

  for n = 1:N
     t = t0 + (n - 1)  h; % Calcular el tiempo actual
     k1 = h  f(t, y(n));
     k2 = h  f(t + h / 2, y(n) + k1 / 2);
     k3 = h  f(t + h / 2, y(n) + k2 / 2);
     k4 = h  f(t + h, y(n) + k3);
     y(n + 1) = y(n) + (k1 + 2  k2 + 2  k3 + k4) / 6; % Fórmula de RK4
```

```
  end
end
```

En este código, f es la función que define la ecuación diferencial, y0 es el valor inicial de la solución, t0 y tf son los límites del tiempo, y h es el tamaño del paso. La función runge_kutta_4 genera un vector y que contiene las aproximaciones de la solución en los tiempos discretos. Para ilustrar la implementación del método de Runge-Kutta en GNU Octave, consideremos la siguiente ecuación diferencial simple:

```
\[
\frac{dy}{dt} = y, \quad y(0) = 1
\]
```

Esta ecuación tiene como solución exacta \(y(t) = e^t\). A continuación, se muestra cómo implementar y comparar la solución aproximada utilizando el método RK4:

```
octave
% Definir la función que describe la ecuación diferencial
function dy = f(t, y)
  dy = y; % Ecuación diferencial
end

% Parámetros del método
y0 = 1;      % Valor inicial
t0 = 0;      % Tiempo inicial
```

```
tf = 5;      % Tiempo final
h = 0.1;     % Tamaño del paso

% Llamar a la función de Runge-Kutta
y_approx = runge_kutta_4(@f, y0, t0, tf, h);

% Generar los tiempos correspondientes
N = (tf - t0) / h;
t_values = t0:h:tf;

% Graficar la solución aproximada y la solución exacta
figure;
plot(t_values, y_approx, 'r-', 'LineWidth', 2); % Solución aproximada
hold on;
plot(t_values, exp(t_values), 'b--', 'LineWidth', 2); % Solución exacta
xlabel('Tiempo t');
ylabel('Solución y');
title('Método de Runge-Kutta: Aproximación vs. Solución Exacta');
legend('Aproximación RK4', 'Solución Exacta');
grid on;
hold off;
```

Este ejemplo no solo muestra la implementación del método RK4, sino que también permite visualizar la eficacia del método al comparar la solución aproximada con la solución exacta. A través de este proceso, los usuarios pueden observar cómo el método de Runge-Kutta puede ser utilizado de

manera efectiva en GNU Octave para resolver ecuaciones diferenciales ordinarias.

Para implementar el método de Runge-Kutta en Python, es fundamental contar con las bibliotecas adecuadas que faciliten tanto el manejo de cálculos numéricos como la visualización de resultados. Las bibliotecas más comunes y recomendadas para este propósito son NumPy y Matplotlib. NumPy proporciona funciones eficientes para trabajar con arreglos y realizar cálculos matemáticos complejos, mientras que Matplotlib permite crear gráficos y visualizaciones de los resultados obtenidos (Downey, 2023). Para instalar estas bibliotecas, se puede utilizar el gestor de paquetes pip. En la terminal o consola de comandos, se pueden ejecutar los siguientes comandos:

```
bash

pip install numpy matplotlib
```

Una vez que se han instalado las bibliotecas, se está listo para proceder con la implementación del método de Runge-Kutta. Existen diferentes variantes del método de Runge-Kutta, siendo el más común el método de cuarto orden. Seguidamente, se presenta una estructura básica en Python para implementar este método. La sintaxis sigue el siguiente formato:

```
python

import numpy as np


def runge_kutta(f, y0, t0, tf, h):
    n = int((tf - t0) / h)  Número de pasos
    t = np.linspace(t0, tf, n + 1)  Array de tiempos
    y = np.zeros(n + 1)  Array para almacenar resultados
    y[0] = y0  Condición inicial


    for i in range(n):
```

```
    k1 = h  f(t[i], y[i])
    k2 = h  f(t[i] + h / 2, y[i] + k1 / 2)
    k3 = h  f(t[i] + h / 2, y[i] + k2 / 2)
    k4 = h  f(t[i] + h, y[i] + k3)
    y[i + 1] = y[i] + (k1 + 2  k2 + 2  k3 + k4) / 6   Actualización del valor

  return t, y
```

En este código, f representa la función que describe la ecuación diferencial, y0 es el valor inicial, t0 y tf son los límites de tiempo, y h es el tamaño del paso. La función devuelve un array de tiempos y otro de valores aproximados de la solución. Para ilustrar la implementación del método de Runge-Kutta en Python, consideremos la ecuación diferencial simple:

```
\[
\frac{dy}{dt} = -2y
\]
```

Con la condición inicial \(y(0) = 1 \). La solución analítica de esta ecuación es \(y(t) = e^{-2t} \). A continuación, se presenta un ejemplo completo utilizando el método de Runge-Kutta:

```
python
import numpy as np
import matplotlib.pyplot as plt

 Definición de la función
```

```
def f(t, y):
    return -2  y

 Parámetros
y0 = 1   Condición inicial
t0 = 0   Tiempo inicial
tf = 5   Tiempo final
h = 0.1   Tamaño del paso

 Implementación del método de Runge-Kutta
t, y = runge_kutta(f, y0, t0, tf, h)

 Solución analítica para comparación
y_analytical = np.exp(-2  t)

 Gráficos
plt.figure(figsize=(10, 5))
plt.plot(t, y, label='Runge-Kutta', marker='o')
plt.plot(t, y_analytical, label='Solución Analítica', linestyle='--')
plt.title('Método de Runge-Kutta vs Solución Analítica')
plt.xlabel('Tiempo')
plt.ylabel('y(t)')
plt.legend()
plt.grid()
plt.show()
```

En este ejemplo, se define la función f, se establecen los parámetros necesarios y se llama a la función runge_kutta para obtener los valores aproximados. Luego se comparan los resultados obtenidos con la solución analítica mediante gráficos, lo que permite visualizar la precisión del método. Con esta implementación, se puede apreciar la efectividad del método de Runge-Kutta para resolver ecuaciones diferenciales ordinarias en Python, destacando su simplicidad y potencia.

Ambas plataformas, GNU Octave y Python, ofrecen potentes capacidades para implementar el método de Runge-Kutta. GNU Octave, con su sintaxis similar a MATLAB, se presenta como una opción accesible para aquellos que ya están familiarizados con este entorno. Por otro lado, Python, con su rica colección de bibliotecas como NumPy y SciPy, proporciona un ecosistema más amplio y flexible, permitiendo realizar análisis más complejos y combinaciones de técnicas. La elección entre uno u otro dependerá del contexto de uso. GNU Octave es ideal para quienes buscan una herramienta específica para la resolución de ecuaciones diferenciales, pese a que Python se adapta mejor a entornos de desarrollo más integrales donde se requiere una mayor variedad de aplicaciones científicas y de ingeniería.

El método de Runge-Kutta y sus variantes continuarán siendo de vital importancia en el ámbito científico y tecnológico. El método de Runge-Kutta no solo es una herramienta clave en la resolución de ecuaciones diferenciales ordinarias, sino que encima se encuentra en el centro de un campo en constante evolución que promete seguir impactando diversas disciplinas en el futuro.

Capítulo II

Álgebra Lineal Numérica con GNU Octave y Python

El álgebra lineal numérica es una rama fundamental de las matemáticas que se centra en el estudio de vectores, matrices y sistemas de ecuaciones lineales, así como en sus aplicaciones en diversos campos como la ingeniería, la física, la economía y la informática. Entre los aspectos más destacados del álgebra lineal numérica es su aplicación en el ámbito de la computación. Hoy en día, se han desarrollado diversas herramientas y lenguajes de programación que facilitan la implementación de algoritmos de álgebra lineal. GNU Octave y Python, en particular, son dos entornos de programación ampliamente utilizados que ofrecen potentes bibliotecas y funciones para realizar operaciones de álgebra lineal de manera eficiente.

En este capítulo, exploraremos los conceptos básicos de álgebra lineal, así como su implementación en GNU Octave y Python. A través de ejemplos prácticos y comparaciones de rendimiento, buscaremos ilustrar cómo estas herramientas pueden ser utilizadas para resolver problemas numéricos, optimizar cálculos y facilitar el análisis de datos. Al final, esperamos proporcionar una comprensión sólida del álgebra lineal numérica y su relevancia en el contexto actual.

2.1 Conceptos básicos de álgebra lineal

El álgebra lineal se ocupa de las propiedades y las relaciones de los vectores, las matrices y los espacios vectoriales, su relevancia se extiende a diversas disciplinas, desde la física y la ingeniería hasta la economía y la estadística. Los vectores son objetos matemáticos que tienen tanto magnitud como dirección. En el contexto del álgebra lineal, un vector se puede representar como una lista ordenada de números, que pueden ser considerados como coordenadas en un espacio n-dimensional (Poole, 2011). Por ejemplo, en el espacio tridimensional, un vector podría representarse como $\mathbf{v} = [x, y, z]$, donde x, y y z son sus componentes.

Por otro lado, las matrices son arreglos bidimensionales de números organizados en filas y columnas. Una matriz se puede denotar como A y puede tener dimensiones $m \times n$, donde m es el número de filas y n es el número de columnas. Las matrices son esenciales para representar sistemas de ecuaciones lineales y para realizar transformaciones lineales en el espacio. Las operaciones básicas en álgebra lineal incluyen la suma y la multiplicación de vectores y matrices.

- Suma de vectores: Dos vectores del mismo tamaño se pueden sumar componente a componente. Si $\mathbf{u} = [u_1, u_2, \ldots, u_n]$ y $\mathbf{v} = [v_1, v_2, \ldots, v_n]$, entonces su suma es:

$$
\mathbf{u} + \mathbf{v} = [u_1 + v_1, u_2 + v_2, \ldots, u_n + v_n]
$$

- Multiplicación de vectores: La multiplicación de un vector por un escalar simplemente implica multiplicar cada componente del vector por ese escalar. En efecto, si c es un escalar, entonces $c \cdot \mathbf{u} = [c \cdot u_1, c \cdot u_2, \ldots, c \cdot u_n]$.

- Multiplicación de matrices: La multiplicación de matrices es más compleja y se basa en el producto de filas por columnas. Si tenemos dos matrices A de tamaño $m \times n$ y B de tamaño $n \times p$, el producto $C = A \cdot B$ será una matriz C de tamaño $m \times p$, donde cada elemento c_{ij} se calcula como:

$$
c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}
$$

El determinante es un escalar asociado a una matriz cuadrada que ofrece información importante sobre la matriz y el sistema de ecuaciones que representa. Para una matriz A de tamaño $n \times n$, el determinante se denota como $\text{det}(A)$ o $|A|$. Las propiedades más relevantes del determinante incluyen:

- Un determinante de cero indica que la matriz es singular, lo que significa que no tiene inversa y el sistema de ecuaciones asociado no tiene solución única.

- Si el determinante es diferente de cero, la matriz es no singular y tiene una solución única.

- El determinante es multiplicativo: $\text{det}(A \cdot B) = \text{det}(A) \cdot \text{det}(B)$.

Estos conceptos básicos del álgebra lineal son esenciales para avanzar en el estudio de temas más complejos y para la aplicación práctica en herramientas computacionales como GNU Octave y Python. GNU Octave es una potente herramienta de software libre que ofrece un entorno de programación similar a MATLAB, facilitando la realización de cálculos numéricos, especialmente en el campo del álgebra lineal. Este software es particularmente apreciado por su accesibilidad y su capacidad para manejar grandes conjuntos de datos, lo que lo convierte en una elección popular entre estudiantes, investigadores y profesionales.

La instalación de GNU Octave es un proceso relativamente sencillo. Está disponible para múltiples sistemas operativos, incluidos Windows, macOS y diversas distribuciones de Linux. Para instalar Octave, se pueden seguir los siguientes pasos:

i. *Descarga:* Visita la página oficial de GNU Octave (https://www.gnu.org/software/octave/) y selecciona la versión adecuada para tu sistema operativo.
ii. *Instalación en Windows:* Ejecuta el archivo .exe descargado y sigue las instrucciones del asistente de instalación. Asegúrate de incluir la opción de instalar los paquetes adicionales si se presentan.

iii. *Instalación en macOS:* Puedes utilizar Homebrew para instalar Octave. Abre la terminal y ejecuta el siguiente comando:

```
bash

brew install octave
```

iv. *Instalación en Linux:* Generalmente, Octave está disponible en los repositorios de las principales distribuciones. Puedes instalarlo utilizando el gestor de paquetes de tu elección. Así, en Ubuntu, puedes ejecutar:

```
bash

sudo apt-get install octave
```

Una vez instalado, se puede abrir GNU Octave desde el menú de aplicaciones o mediante la línea de comandos e incluye una serie de funciones incorporadas que facilitan la realización de operaciones de álgebra lineal. Algunas de las funciones más utilizadas son:

- Vectores y matrices: Puedes crear vectores y matrices utilizando la notación de corchetes. En este caso, un vector fila se concreta como v = [1, 2, 3], mientras que una matriz se puede definir como A = [1, 2; 3, 4].

- Suma y multiplicación: La suma de matrices y vectores se realiza utilizando el operador +, en tanto que la multiplicación matricial se logra con el operador . En particular, si definimos B = [5, 6; 7, 8], podemos sumar A + B o multiplicar A B.

- Determinantes: El cálculo del determinante de una matriz se realiza con la función det(). Para ilustrar, det(A) devolverá el determinante de la matriz A.

- Inversa de una matriz: La inversa de una matriz se puede calcular utilizando la función inv(). Si A es invertible, inv(A) devolverá su matriz inversa.

Para ilustrar el uso de GNU Octave en álgebra lineal, consideremos algunos ejemplos prácticos:

i. Definición de una matriz y su determinante:

```
octave
```

```
A = [1, 2; 3, 4];
det_A = det(A);
printf("El determinante de A es: %f\n", det_A);
```

ii. Suma y multiplicación de matrices:

```
octave
B = [5, 6; 7, 8];
C = A + B;  % Suma
D = A  B;  % Multiplicación
printf("La suma de A y B es:\n");
disp(C);
printf("La multiplicación de A y B es:\n");
disp(D)
```

iii. Cálculo de la inversa:

```
octave
inv_A = inv(A);
printf("La inversa de A es:\n");
disp(inv_A);
```

Estos ejemplos muestran cómo GNU Octave permite realizar cálculos complejos de álgebra lineal de manera simple y eficiente. La facilidad de uso y la sintaxis intuitiva de Octave lo convierten en una herramienta valiosa para quienes desean profundizar en el álgebra lineal numérica. La implementación del álgebra lineal en Python se ha vuelto cada vez más popular, especialmente debido a la disponibilidad de bibliotecas poderosas y eficientes que facilitan el trabajo con vectores y matrices. Dos de las bibliotecas más destacadas en este ámbito son NumPy y SciPy, que ofrecen herramientas robustas para realizar operaciones matemáticas complejas.

NumPy es una biblioteca fundamental para la computación científica en Python. Proporciona un potente objeto de matriz multidimensional que permite realizar operaciones matemáticas de manera eficiente. Con su sintaxis intuitiva y su capacidad para trabajar con grandes conjuntos de datos, NumPy es ideal para implementar álgebra lineal. Por otro lado, SciPy se construye sobre NumPy y añade funcionalidades adicionales para realizar cálculos más avanzados, como la resolución de sistemas de ecuaciones, la descomposición de matrices y la optimización (Riyantoko et al., 2025). Para instalar estas bibliotecas, basta con utilizar el gestor de paquetes de Python, pip. En efecto, se pueden instalar ejecutando los siguientes comandos en la terminal:

```
bash
pip install numpy
pip install scipy
```

Una vez instaladas las bibliotecas, podemos comenzar a realizar operaciones con matrices. En Python, las matrices se representan como arreglos de NumPy. He aquí operaciones comunes que se pueden llevar a cabo:

i. *Creación de matrices:* Para crear una matriz, utilizamos la función np.array(). Hay que hacer notar:

```
python
import numpy as np
 Crear una matriz 2x2
matriz = np.array([[1, 2], [3, 4]])
print(matriz)
```

ii. *Suma de matrices:* La suma de matrices de igual tamaño se realiza utilizando el operador +:

```
python
matriz_a = np.array([[1, 2], [3, 4]])
matriz_b = np.array([[5, 6], [7, 8]])
suma = matriz_a + matriz_b
print(suma)
```

iii. *Multiplicación de matrices:* Para multiplicar matrices, se utiliza la función np.dot() o el operador @:

```
python
producto = np.dot(matriz_a, matriz_b)
 O alternativamente
producto = matriz_a @ matriz_b
print(producto)
```

iv. *Determinantes y otras propiedades:* Para calcular el determinante de una matriz, se utiliza la función np.linalg.det():

```
python
determinante = np.linalg.det(matriz_a)
print(determinante)
```

No obstante tanto GNU Octave como Python son herramientas poderosas para el álgebra lineal, existen diferencias en su rendimiento y en la facilidad de uso. Octave, al estar diseñado específicamente para cálculos numéricos, puede ofrecer un rendimiento superior en algunas operaciones básicas. Sin embargo, Python, con bibliotecas como NumPy y SciPy, proporciona una flexibilidad y una integración con otros entornos que lo hacen particularmente atractivo para desarrolladores y científicos de datos.

En pruebas de rendimiento, Python tiende a ser más rápido en operaciones complejas y en el manejo de grandes volúmenes de datos, gracias a su capacidad para optimizar el uso de memoria y su diseño orientado a objetos. Por otro lado, Octave puede ser más accesible para quienes están familiarizados con MATLAB, debido a su sintaxis similar. GNU Octave, con su enfoque en la facilidad de uso y su similitud con MATLAB, es una opción excelente para quienes buscan un entorno dedicado al álgebra lineal y a la computación numérica (Lopez y Riasco, 2024). Su capacidad para manejar matrices y realizar cálculos complejos de manera intuitiva lo convierte en una herramienta valiosa para estudiantes y profesionales por igual.

Por otro lado, Python, con bibliotecas como NumPy y SciPy, no solo proporciona potentes funcionalidades para álgebra lineal, sino que también se integra fácilmente con otras herramientas y tecnologías, lo que lo hace ideal para proyectos más amplios que requieren análisis de datos, aprendizaje automático y desarrollo de software.

La continua evolución de las bibliotecas de Python y el incremento de su popularidad en la comunidad científica sugieren que Python seguirá siendo una opción predominante para el análisis numérico. Por su parte, GNU Octave podría beneficiarse de mejoras en su interfaz y en la interoperabilidad con otros lenguajes de programación, lo que podría aumentar su atractivo entre los usuarios que buscan un entorno más especializado.

Además, el auge de la computación en la nube y el acceso a hardware de alto rendimiento permitirá a los investigadores y desarrolladores realizar cálculos más complejos y manejar conjuntos de datos más grandes, lo que seguramente impulsará la evolución de las herramientas de álgebra lineal. La combinación de técnicas de álgebra lineal con algoritmos de inteligencia

artificial y aprendizaje automático incluso abre nuevas perspectivas para resolver problemas complejos en diversas disciplinas.

Tanto GNU Octave como Python ofrecen soluciones robustas para el álgebra lineal numérica, y su desarrollo continuo promete ampliar las posibilidades para los usuarios. La elección entre estos entornos dependerá de las necesidades individuales y del contexto específico de cada proyecto, pero lo que es indudable es que el dominio de estas herramientas seguirá siendo un activo valioso en el mundo académico y profesional.

2.2 Eliminación de Gauss y Gauss-Jordan: Técnicas Esenciales para Resolver Problemas Ideales

La eliminación de Gauss, también conocida como eliminación gaussiana, se centra en transformar una matriz en su forma escalonada, permitiendo resolver sistemas de ecuaciones mediante un proceso de eliminación sucesiva. Por otro lado, la eliminación de Gauss-Jordan lleva este método un paso más allá, reduciendo una matriz a su forma escalonada reducida, lo que proporciona no solo las soluciones del sistema, sino de igual modo una representación más clara de las relaciones entre las variables (Guanga et al., 2019). La eliminación de Gauss es un método fundamental en álgebra lineal que permite resolver sistemas de ecuaciones lineales. Su desarrollo se basa en la idea de transformar una matriz de coeficientes en una forma más sencilla, lo que facilita la obtención de soluciones.

La eliminación de Gauss, también conocida como eliminación gaussiana, es un algoritmo que transforma una matriz en su forma escalonada, lo que permite resolver sistemas de ecuaciones lineales de manera sistemática. El propósito principal de este método es simplificar el sistema de ecuaciones para que se puedan identificar las soluciones de manera clara y directa. Este proceso es esencial no solo en matemáticas puras, sino también en diversas áreas aplicadas, como la física, la ingeniería y la economía, donde los sistemas de ecuaciones lineales son frecuentes. El proceso de eliminación de Gauss se puede dividir en tres etapas principales:

i. *Formación de la matriz aumentada:* Se comienza formando la matriz aumentada del sistema de ecuaciones a resolver. Esta matriz

combina los coeficientes de las variables y los términos independientes en una sola matriz.

ii. *Eliminación hacia adelante:* En esta etapa, se aplican operaciones elementales sobre las filas de la matriz para transformar la parte inferior de la matriz en ceros. Se selecciona un pivote (usualmente el primer elemento no nulo de la primera fila) y se utilizan combinaciones lineales de las filas para eliminar los coeficientes debajo de este pivote.

iii. *Sustitución hacia atrás:* Una vez que la matriz está en forma escalonada, se procede a realizar la sustitución hacia atrás para encontrar los valores de las variables. Comenzando desde la última fila, se resuelve cada variable en términos de las que ya se han encontrado. Este proceso garantiza que, si existe una solución, se obtendrá de manera eficiente, para ilustrar el proceso de eliminación de Gauss, consideremos el siguiente sistema de ecuaciones lineales:

```
\[
\begin{align}
2x + 3y + z &= 1 \\
4x + y - 2z &= -2 \\
-2x + 5y + 3z &= 3
\end{align}
\]
```

i. Formación de la matriz aumentada:

```
\[
\begin{pmatrix}
```

```
2 & 3 & 1 & | & 1 \\
4 & 1 & -2 & | & -2 \\
-2 & 5 & 3 & | & 3
\end{pmatrix}
\]
```

ii. *Eliminación hacia adelante:* Aplicamos operaciones para transformar la matriz:

- Multiplicamos la primera fila por 2 y restamos de la segunda fila.

- Multiplicamos la primera fila por -1 y sumamos a la tercera fila.

Esto lleva a una matriz escalonada como:

```
\[
\begin{pmatrix}
2 & 3 & 1 & | & 1 \\
0 & -5 & -4 & | & -4 \\
0 & 8 & 5 & | & 5
\end{pmatrix}
\]
```

Continuamos el proceso hasta que todas las filas estén en la forma deseada.

iii. *Sustitución hacia atrás:* Con la matriz escalonada, se resuelven las variables comenzando desde la última fila hacia la primera. Este ejemplo demuestra cómo la eliminación de Gauss simplifica el proceso de resolución de sistemas de ecuaciones lineales, permitiendo a los estudiantes y profesionales abordar problemas complejos de manera más accesible.

La eliminación de Gauss-Jordan es una extensión del método de eliminación de Gauss, que se utiliza para resolver sistemas de ecuaciones lineales. Este método no solo busca transformar una matriz en su forma escalonada, sino que incluso la lleva a una forma reducida, conocida como forma escalonada reducida por filas (FERF). Esta transformación permite obtener soluciones de manera más directa y eficiente, facilitando la identificación de las variables y su relación con las ecuaciones del sistema.

La principal diferencia entre los métodos de eliminación de Gauss y Gauss-Jordan radica en el tipo de matriz resultante; en el método de eliminación de Gauss, el objetivo es lograr una forma escalonada, donde los elementos por debajo de la diagonal principal son ceros. Sin embargo, en Gauss-Jordan, se lleva el proceso un paso más allá: se busca que todos los elementos en la columna de cada pivote sean cero, excepto el pivote mismo, que se convierte en 1, esto proporciona una forma más simplificada que permite leer las soluciones de las variables directamente (Espinosa et al., 2016).

Otra diferencia clave es que, mientras que la eliminación de Gauss generalmente requiere una etapa adicional para deshacer la eliminación y obtener los valores de las variables, la eliminación de Gauss-Jordan proporciona las soluciones en una sola etapa, lo que la hace más eficiente en términos de tiempo y recursos. El proceso de eliminación de Gauss-Jordan implica los siguientes pasos:

i. *Formación de la matriz aumentada:* Se comienza con la matriz aumentada del sistema de ecuaciones, que incluye tanto los coeficientes de las variables como los términos independientes.
ii. *Aplicación de operaciones elementales:* Se realizan operaciones elementales sobre las filas de la matriz, que incluyen:

- Intercambiar dos filas.

- Multiplicar una fila por un escalar distinto de cero.

- Sumar o restar un múltiplo de una fila a otra fila.

iii. *Transformación a FERF:* Se busca convertir la matriz en su forma escalonada reducida por filas. Esto significa que se debe conseguir que cada

pivote sea 1 y que todos los elementos en la columna del pivote sean ceros, a excepción del propio pivote.

iv. *Lectura de las soluciones:* Una vez que la matriz está en FERF, las soluciones del sistema pueden leerse directamente, lo que facilita la identificación de las variables y sus valores. Para ilustrar la aplicación de la eliminación de Gauss-Jordan, consideremos un sistema simple de dos ecuaciones con dos incógnitas:

1. \(x + 2y = 8 \)
2. \(2x + y = 7 \)

La matriz aumentada de este sistema es:

\[
\begin{pmatrix}
1 & 2 & | & 8 \\
2 & 1 & | & 7
\end{pmatrix}
\]

Aplicando la eliminación de Gauss-Jordan, el primer paso sería transformar la segunda fila para eliminar el 2 en la primera columna. Esto se puede lograr restando 2 veces la primera fila de la segunda fila:

\[
\begin{pmatrix}
1 & 2 & | & 8 \\

```
0 & -3 & | & -9
\end{pmatrix}
\]
```

Luego, se puede simplificar la segunda fila dividiendo por -3:

```
\[
\begin{pmatrix}
1 & 2 & | & 8 \\
0 & 1 & | & 3
\end{pmatrix}
\]
```

Lurgo, eliminamos el 2 en la primera fila de la segunda columna restando 2 veces la segunda fila de la primera fila:

```
\[
\begin{pmatrix}
1 & 0 & | & 2 \\
0 & 1 & | & 3
\end{pmatrix}
\]
```

De esta forma, podemos leer que \(x = 2 \) y \(y = 3 \), resolviendo el sistema de manera eficiente con la eliminación de Gauss-Jordan, este enfoque no solo simplifica la resolución de sistemas lineales, sino que encima sienta

las bases para aplicaciones más complejas en matemáticas y ciencias aplicadas. Con la comprensión de este método, se abre un abanico de posibilidades para resolver problemas matemáticos de manera efectiva y rápida. Las eliminaciones de Gauss y Gauss-Jordan son técnicas fundamentales en el ámbito de las matemáticas aplicadas, especialmente en la resolución de sistemas de ecuaciones lineales. Estas metodologías no solo son cruciales en la teoría matemática, sino que también tienen numerosas aplicaciones prácticas en diversas disciplinas.

Entre las aplicaciones más directas de las eliminaciones de Gauss y Gauss-Jordan está la resolución de sistemas de ecuaciones lineales, estas técnicas permiten transformar un sistema de ecuaciones en una forma más manejable, facilitando la búsqueda de soluciones (Espinosa et al., 2016). Así, en situaciones donde se necesita determinar los valores de variables en un sistema de ecuaciones que representan un modelo matemático, estas herramientas permiten simplificar el sistema a una forma escalonada o escalonada reducida, desde donde las soluciones pueden ser fácilmente identificadas. Esto es especialmente útil en campos como la economía, donde se modelan interacciones entre diferentes variables económicas.

En el campo de la ingeniería, las eliminaciones de Gauss y Gauss-Jordan son esenciales para resolver problemas relacionados con circuitos eléctricos, estructuras y sistemas dinámicos. En particular, en el análisis de circuitos, se pueden utilizar estas técnicas para resolver sistemas de ecuaciones que describen las relaciones entre voltajes y corrientes en un circuito. De manera similar, en ciencias como la física y la química, estas metodologías se aplican para resolver sistemas que describen fenómenos naturales, desde la cinética de reacciones químicas hasta el equilibrio de fuerzas en un sistema mecánico. La capacidad de manejar múltiples ecuaciones simultáneamente es invaluable en estos contextos.

En la actualidad, se han desarrollado diversas herramientas y software que implementan las eliminaciones de Gauss y Gauss-Jordan, permitiendo a los usuarios resolver sistemas de ecuaciones de manera eficiente y precisa. Programas como MATLAB, Mathematica y Python (con bibliotecas como NumPy) ofrecen funciones que automatizan el proceso de eliminación, facilitando la resolución de problemas complejos sin la necesidad de realizar

cálculos manuales extensos. Estas herramientas no solo ahorran tiempo, sino que de igual modo minimizan errores, mejorando la precisión de los resultados obtenidos.

Las eliminaciones de Gauss y Gauss-Jordan son técnicas poderosas con aplicaciones que se extienden a múltiples campos, desde la resolución de sistemas de ecuaciones lineales hasta su uso en ingeniería y ciencias. La disponibilidad de software especializado ha ampliado aún más su utilidad, permitiendo que profesionales y estudiantes resuelvan problemas complejos de manera eficiente.

La eliminación de Gauss se centra en transformar una matriz a su forma escalonada, facilitando la resolución de sistemas lineales mediante sustitución regresiva. Este método es particularmente útil para resolver sistemas en los que se busca determinar una o más variables a partir de un conjunto de ecuaciones. Por otro lado, la eliminación de Gauss-Jordan lleva este proceso un paso más allá, permitiendo obtener la forma reducida por filas de una matriz, lo que proporciona soluciones directas y únicas para los sistemas de ecuaciones lineales, así como también para encontrar inversas de matrices cuando estas existen.

Las aplicaciones prácticas de estas técnicas son vastas y se extienden a campos como la ingeniería, la física y la economía, donde la modelización de problemas complejos mediante sistemas de ecuaciones es común. Además, en la era digital, el uso de software y herramientas computacionales para realizar estas eliminaciones ha simplificado enormemente el proceso, permitiendo a los profesionales y estudiantes abordar problemas que antes serían tediosos y propensos a errores manuales.

Tanto la eliminación de Gauss como la de Gauss-Jordan son herramientas esenciales en el arsenal de cualquier estudiante o profesional que trabaje con álgebra lineal. Su comprensión y dominio no solo son cruciales para resolver problemas matemáticos, sino que incluso abren la puerta a un mejor entendimiento de conceptos más avanzados en matemáticas y sus aplicaciones en el mundo real.

2.3 Eliminación Canónica de Gauss y Pivoteo: Implementaciones en GNU Octave y Python

La eliminación canónica de Gauss es un método fundamental en álgebra lineal, utilizado para resolver sistemas de ecuaciones lineales. Este procedimiento transforma un sistema de ecuaciones en una forma más sencilla, conocida como forma escalonada, lo que permite determinar las soluciones de manera más eficiente. La técnica implica manipular matrices mediante operaciones elementales, que incluyen la eliminación de variables y la sustitución hacia atrás.

La eliminación canónica de Gauss, que se conoce como eliminación de Gauss-Jordan cuando se lleva a cabo hasta obtener la forma reducida, consiste en aplicar una serie de transformaciones a una matriz aumentada que representa un sistema de ecuaciones lineales. El objetivo es obtener ceros debajo de la diagonal principal, facilitando la obtención de soluciones del sistema. Este método es útil no solo para resolver ecuaciones, sino también para calcular determinantes y encontrar inversas de matrices.

El pivoteo es una técnica complementaria a la eliminación canónica de Gauss que mejora la estabilidad numérica del proceso. Consiste en seleccionar adecuadamente los elementos de la matriz que se utilizan como pivotes, es decir, los elementos en la diagonal principal durante el proceso de eliminación, al elegir el pivote más grande en valor absoluto de cada columna, se minimizan los errores de redondeo y se mejora la precisión del cálculo (Strang, 1980). Esto es especialmente decisivo cuando se trabaja con matrices que tienen elementos muy pequeños o en casos de singularidad. La eliminación canónica de Gauss implica transformar un sistema de ecuaciones lineales en una forma escalonada, lo que permite resolverlo de manera más sencilla. En GNU Octave, la sintaxis para implementar la eliminación de Gauss con pivoteo se puede estructurar de la siguiente manera:

```
octave
function [A] = gauss_elimination(A)
  [m, n] = size(A);
```

```
    for k = 1:min(m, n)
        % Pivoteo: encontrar la fila con el mayor valor absoluto en la columna k
        [~, maxIndex] = max(abs(A(k:m, k)));
        maxIndex = maxIndex + k - 1;

        % Intercambiar filas
        if maxIndex != k
            A([k, maxIndex], :) = A([maxIndex, k], :);
        end

        % Eliminación
        for i = k+1:m
            factor = A(i, k) / A(k, k);
            A(i, :) = A(i, :) - factor  A(k, :);
        end
    end
end
```

En este código, A es la matriz aumentada que contiene tanto los coeficientes del sistema como los términos independientes. La función realiza el pivoteo para asegurar la estabilidad numérica y modifica la matriz para obtener la forma escalonada. Para ilustrar la implementación de la eliminación canónica de Gauss con pivoteo, consideremos el siguiente sistema de ecuaciones lineales:

```
\[
\begin{align}
2x + 3y + z &= 1 \\
4x + y + 2z &= 2 \\
3x + 2y + 3z &= 3
\end{align}
\]
```

Podemos representar este sistema en forma de matriz aumentada A:

```
octave
A = [2, 3, 1, 1;
    4, 1, 2, 2;
    3, 2, 3, 3];

% Aplicar la eliminación de Gauss
A_echelon = gauss_elimination(A);

% Mostrar la matriz resultante
disp(A_echelon);
```

Al ejecutar este código, obtendrás la forma escalonada de la matriz, que podrás usar para resolver el sistema de ecuaciones de manera más sencilla. Este ejemplo práctico demuestra la implementación de la eliminación canónica de Gauss y resalta la importancia del pivoteo para mejorar la precisión y estabilidad del método. Con esta base en GNU Octave, ahora

estamos listos para pasar a la implementación de la eliminación canónica de Gauss en Python, donde utilizaremos bibliotecas como NumPy y SciPy para lograr un enfoque similar.

Python se ha convertido en uno de los lenguajes de programación más populares para la ciencia de datos y el cálculo numérico, gracias a su sintaxis sencilla y a sus potentes bibliotecas. Para llevar a cabo la implementación de la eliminación canónica de Gauss en Python, primero necesitamos instalar las bibliotecas que nos ayudarán en nuestros cálculos. NumPy es fundamental para el manejo de arreglos y operaciones matemáticas, mientras que SciPy proporciona funciones adicionales para optimización y resolución de problemas científicos. Para instalar estas bibliotecas, utilizaremos pip, el gestor de paquetes de Python. Ejecuta el siguiente comando en la terminal o en un entorno de desarrollo integrado (IDE):

```
bash
pip install numpy scipy
```

La eliminación canónica de Gauss se puede implementar de manera efectiva utilizando NumPy, este código resolverá un sistema de ecuaciones lineales de la forma \(Ax = b\).

```
python
import numpy as np

def gauss_elimination(A, b):
    n = len(b)
     Ampliar la matriz A con el vector b
    Ab = np.hstack([A, b.reshape(-1, 1)])
```

```
    Proceso de eliminación
    for i in range(n):
        Hacer que la fila i tenga un 1 en la diagonal
        Ab[i] = Ab[i] / Ab[i, i]
        for j in range(i + 1, n):
            Ab[j] = Ab[j] - Ab[i]  Ab[j, i]

    Retroceso para resolver el sistema
    x = np.zeros(n)
    for i in range(n - 1, -1, -1):
        x[i] = Ab[i, -1] - np.dot(Ab[i, i + 1:n], x[i + 1:n])

    return x
```

En este código, comenzamos ampliando la matriz \(A\) con el vector \(b\) para formar la matriz aumentada \(Ab\). Luego, realizamos el proceso de eliminación, convirtiendo la matriz en una forma escalonada y aplicamos el método de retroceso para encontrar la solución del sistema. El pivoteo es un paso excepcional para mejorar la estabilidad numérica del algoritmo. En seguida, se presenta una implementación de la eliminación de Gauss con pivoteo parcial:

```
python
def gauss_elimination_with_pivoting(A, b):
    n = len(b)
    Ab = np.hstack([A, b.reshape(-1, 1)])
```

```
    for i in range(n):
        Pivoteo parcial
        max_row = np.argmax(np.abs(Ab[i:n, i])) + i
        Ab[[i, max_row]] = Ab[[max_row, i]]   Intercambiar filas

        Normalizar la fila actual
        Ab[i] = Ab[i] / Ab[i, i]
        for j in range(i + 1, n):
            Ab[j] = Ab[j] - Ab[i]  Ab[j, i]

    x = np.zeros(n)
    for i in range(n - 1, -1, -1):
        x[i] = Ab[i, -1] - np.dot(Ab[i, i + 1:n], x[i + 1:n])

    return x
```

En esta versión, antes de realizar la eliminación, buscamos la fila con el valor absoluto más grande en la columna actual y la intercambiamos con la fila actual. Esto ayuda a evitar problemas de división por cero y mejora la precisión del resultado. Para ilustrar la funcionalidad de nuestra implementación, consideremos el siguiente sistema de ecuaciones:

```
\[
\begin{align}
2x + 3y + z &= 1 \\
4x + y + 2z &= 2 \\
```

```
3x + 2y + 3z &= 3
\end{align}
\]
```

La matriz \(A\) y el vector \(b\) se definen de la siguiente manera:

```
python
A = np.array([[2, 3, 1], [4, 1, 2], [3, 2, 3]], dtype=float)
b = np.array([1, 2, 3], dtype=float)

 Usando la función con pivoteo
solution = gauss_elimination_with_pivoting(A, b)
print("La solución del sistema es:", solution)
```

Al ejecutar este código, obtendremos la solución del sistema de ecuaciones, destacando la efectividad de Python y sus bibliotecas para realizar cálculos numéricos complejos de forma eficiente. GNU Octave es un entorno de programación diseñado específicamente para cálculos numéricos, lo que lo convierte en una opción atractiva para quienes trabajan en matemáticas, ingeniería y ciencias. Entre sus ventajas se encuentran:

i. *Facilidad de uso:* La sintaxis de Octave es similar a la de MATLAB, lo que permite a los nuevos usuarios comenzar a trabajar rápidamente.
ii. *Entorno gráfico:* Octave ofrece un entorno gráfico intuitivo que permite visualizar datos y resultados de manera más sencilla, ideal para quienes prefieren una interfaz visual.
iii. *Soporte para funciones matemáticas:* Incluye una amplia gama de funciones matemáticas predefinidas, lo que reduce la necesidad de implementar soluciones desde cero.

iv. *Código abierto:* Como software libre, GNU Octave permite a los usuarios modificar y distribuir el código según sus necesidades, fomentando la colaboración.

Python se ha consolidado como uno de los lenguajes de programación más versátiles. Sus ventajas incluyen:

i. *Versatilidad:* A diferencia de Octave, Python no está limitado a cálculos numéricos y puede ser utilizado en una amplia variedad de aplicaciones.
ii. *Ecosistema de bibliotecas:* Python cuenta con un rico ecosistema de bibliotecas, como NumPy y SciPy, que facilitan la implementación de algoritmos complejos.
iii. *Comunidad activa:* La comunidad de usuarios y desarrolladores de Python es enorme y activa, lo que se traduce en abundantes recursos y foros de discusión.
iv. *Interoperabilidad:* Python se integra fácilmente con otros lenguajes y tecnologías, lo que permite maximizar la eficiencia en proyectos multidisciplinarios.

La elección entre GNU Octave y Python puede depender de varios factores, incluyendo el contexto del proyecto y las preferencias personales del usuario. Algunos casos de uso recomendados son:

- GNU Octave: Ideal para estudiantes y profesionales que se centran en la educación y la investigación en matemáticas y ciencias aplicadas.

- Python: Perfecto para proyectos que requieren un enfoque más amplio y flexible, siendo la opción preferida en análisis de datos y machine learning.

La elección entre GNU Octave y Python depende de las necesidades específicas del usuario y el contexto de aplicación. GNU Octave se presenta como una opción ideal para quienes prefieren un entorno similar a MATLAB, mientras que Python, con sus potentes bibliotecas, es ideal para una amplia gama de aplicaciones (Companioni et al., 2012). La integración de técnicas de optimización y análisis numérico en el contexto de la eliminación canónica de Gauss promete abrir nuevas avenidas de investigación y aplicación en diversas disciplinas.

Capítulo III

Diferenciación numérica

La diferenciación numérica es una herramienta fundamental en el análisis matemático y la computación científica, que permite calcular aproximaciones de las derivadas de funciones a partir de un conjunto discreto de puntos. A diferencia de la diferenciación analítica, que se basa en fórmulas exactas y propiedades de funciones continuas, la diferenciación numérica se enfoca en la evaluación de funciones en puntos específicos. Esto resulta esencial cuando se trabaja con datos experimentales o en situaciones donde las funciones no son fácilmente diferenciables de manera analítica.

La diferenciación numérica se refiere al proceso de estimar la derivada de una función en un punto específico utilizando valores de la función en puntos cercanos. Este método es particularmente útil cuando solo se dispone de datos discretos o cuando la forma de la función es compleja y no se puede derivar fácilmente. Existen diversos métodos para llevar a cabo la diferenciación numérica, que pueden categorizarse en aproximaciones de orden superior y técnicas que utilizan diferencias finitas.

La diferenciación numérica tiene una amplia gama de aplicaciones en diversas disciplinas, incluyendo la ingeniería, la física, la economía y la biología. En sí, se utiliza en la modelización de sistemas dinámicos, donde es trascendental conocer la tasa de cambio de ciertas variables en función del tiempo. Además, en el análisis de datos experimentales, la diferenciación numérica permite a los investigadores identificar tendencias y comportamientos en los datos, ayudando en la toma de decisiones informadas. En el ámbito financiero, se aplica en la valoración de opciones y en la evaluación de riesgos, donde se requiere calcular derivadas para entender cómo los cambios en ciertos parámetros afectan a los resultados.

La diferenciación numérica no es un concepto nuevo; sus raíces se remontan a los inicios del cálculo en el siglo XVII. Matemáticos como Isaac Newton y Gottfried Wilhelm Leibniz sentaron las bases del cálculo diferencial, si bien sus enfoques eran principalmente analíticos. Sin embargo,

investigadores como John von Neumann y el equipo de la ENIAC comenzaron a explorar técnicas de diferenciación que podían ser implementadas en computadoras, lo que llevó al desarrollo de algoritmos y métodos que son ampliamente utilizados hoy en día. La diferenciación numérica continúa evolucionando, integrándose con nuevas tecnologías y métodos de análisis de datos, lo que la convierte en una herramienta indispensable para los científicos e ingenieros modernos.

3.1 Polinomios de interpolación de Newton

La interpolación es un método matemático utilizado para estimar valores desconocidos dentro del rango de un conjunto discreto de puntos de datos. A través de esta técnica, se busca construir una función que pase exactamente por cada uno de los puntos dados, permitiendo así realizar predicciones o análisis sobre puntos no medidos (Pochulu, 2018). La interpolación se aplica en diversas áreas, como la ingeniería, la economía y la ciencia, donde los datos pueden ser escasos o difíciles de obtener. Dentro de las diversas técnicas de interpolación, el método de Newton es uno de los más utilizados debido a su eficiencia y facilidad para manejar conjuntos de datos.

El polinomio de interpolación de Newton se construye a partir de un conjunto de puntos \((x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\). La forma general del polinomio se expresa como:

\[
P_n(x) = a_0 + a_1 (x - x_0) + a_2 (x - x_0)(x - x_1) + \ldots + a_n (x - x_0)(x - x_1) \cdots (x - x_{n-1}),
\]

donde los coeficientes \(a_0, a_1, \ldots, a_n\) son conocidos como coeficientes de Newton. Estos coeficientes se obtienen a través de las diferencias divididas, que son una forma eficiente de calcular las tasas de cambio de los valores de \(y\) con respecto a los valores de \(x\). La primera diferencia dividida se precisa como:

\[
f[x_i] = y_i,
\]

y las diferencias sucesivas se calculan mediante la fórmula:

\[
f[x_i, x_j] = \frac{f[x_j] - f[x_i]}{x_j - x_i},
\]

y así sucesivamente para \(f[x_i, x_j, x_k]\), etc. Este procedimiento nos permite construir el polinomio de forma iterativa, comenzando desde los valores de \(y\) y utilizando las diferencias divididas para obtener los coeficientes. Los polinomios de interpolación de Newton ofrecen varias ventajas frente a otros métodos de interpolación. Entre las más destacadas se incluyen:

i. *Eficiencia en la adición de puntos:* Entre las principales ventajas de los polinomios de Newton es que, si se desea añadir un nuevo punto al conjunto de datos, no es necesario recalcular todo el polinomio desde cero. En su lugar, bastará con calcular las diferencias divididas adicionales, lo que ahorra tiempo y esfuerzo computacional.
ii. *Flexibilidad:* Este método permite trabajar con conjuntos de datos de diferente tamaño y densidad, lo que lo hace muy versátil en aplicaciones prácticas.
iii. *Estabilidad numérica:* Aunque la interpolación polinómica puede presentar problemas de oscilación en intervalos grandes (fenómeno de Runge), la forma de Newton, que utiliza diferencias divididas, tiende a ser más estable y menos propensa a errores numéricos en

comparación con otros métodos, como la interpolación de Lagrange.

iv. *Facilidad de derivación:* Los polinomios de Newton son particularmente útiles cuando se necesita calcular derivadas, ya que su forma permite derivar fácilmente el polinomio en cualquier punto sin necesidad de reconfigurarlo.

La diferenciación de polinomios de interpolación es una herramienta fundamental en el análisis numérico y en la resolución de problemas matemáticos que involucran tasas de cambio y pendientes, este proceso permite obtener derivadas de funciones aproximadas a partir de un conjunto de puntos discretos, lo que es especialmente útil en situaciones donde la función original no está explícitamente definida (Ortega y Simental, 2021). La diferenciación de polinomios de interpolación se puede realizar de diversas maneras, siendo las más comunes la diferenciación analítica y la diferenciación numérica.

i. *Diferenciación analítica:* Dado un polinomio de interpolación expresado en forma de Newton, la derivada se puede calcular directamente aplicando las reglas estándar de derivación. En particular, si tenemos un polinomio \(P(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \ldots + a_n(x - x_0)(x - x_1)\cdots(x - x_{n-1}) \), la derivada \(P'(x) \) se puede obtener fácilmente aplicando la derivada a cada término del polinomio.

ii. *Diferenciación numérica:* En situaciones donde la función no está definida de manera explícita o se tiene un polinomio de alto grado, se pueden emplear métodos numéricos para calcular la derivada. Entre los métodos más utilizados es la diferencia dividida, que permite estimar la derivada de una función en un punto utilizando valores de la función en puntos cercanos. Este enfoque es particularmente útil en el contexto de la interpolación, ya que se basa en los valores de la función en los nodos de interpolación. de polinomios de interpolación, existen varios errores comunes que pueden comprometer la precisión de los resultados:

i. *Elección inadecuada de nodos:* La selección de los puntos de interpolación es esencial. Si los nodos están demasiado próximos

entre sí o no son representativos de la función, la aproximación puede resultar poco precisa.

ii. *Polinomios de alto grado:* Los polinomios de interpolación de grado elevado pueden producir oscilaciones indeseadas (fenómeno de Runge) que afectan la precisión de la diferenciación. Es fundamental considerar el grado del polinomio y, en algunos casos, optar por interpolaciones de menor grado o métodos de interpolación locales.

iii. *Errores de redondeo:* En cálculos numéricos, los errores de redondeo pueden acumularse, especialmente cuando se realizan operaciones con números muy pequeños o muy grandes. Estos errores pueden ser particularmente significativos en la diferenciación de polinomios.

Para ilustrar la diferenciación de polinomios de interpolación, consideremos el siguiente ejemplo:

Supongamos que tenemos un conjunto de puntos $(x_0, y_0), (x_1, y_1), (x_2, y_2)$ donde $y_i = f(x_i)$. A partir de estos puntos, construimos el polinomio de interpolación de Newton:

$$P(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1)$$

Para encontrar la derivada en un punto específico, aplicamos la diferenciación analítica:

$$P'(x) = a_1 + a_2[(x - x_0) + (x - x_1)]$$

Si deseamos calcular numéricamente la derivada en un punto x_0 utilizando diferencias divididas, podríamos estimar:

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}$$

donde h es un pequeño incremento en x.

La diferenciación de polinomios de interpolación es un procedimiento poderoso que combina la teoría de interpolación con la práctica de la diferenciación. Con el conocimiento de los métodos adecuados y la atención a los errores comunes, los matemáticos e ingenieros pueden aplicar estas técnicas con eficacia en una amplia gama de problemas. La capacidad de aproximar derivadas de funciones a partir de datos discretos no solo es vital en el contexto académico, sino que encima tiene un impacto significativo en la toma de decisiones en la vida real.

Los polinomios de interpolación de Newton, como hemos visto, ofrecen una forma eficiente y efectiva de modelar funciones a partir de un conjunto de puntos. Su construcción, basada en diferencias divididas, permite no solo la aproximación de la función original, sino de igual modo una derivación precisa de sus características. Las ventajas que presentan, como la facilidad de actualización y la estabilidad numérica, los convierten en una herramienta valiosa en la diferenciación numérica.

Es importante recordar que, si bien la diferenciación de polinomios de interpolación puede ser un proceso poderoso, incluso conlleva sus desafíos. Los errores comunes, como la elección inadecuada de puntos de interpolación o la mala interpretación de los resultados, pueden llevar a conclusiones erróneas. Por ello, es esencial aplicar estos métodos con cuidado y realizar un análisis crítico de los resultados obtenidos.

3.2 Diferenciación numérica: Uso del desarrollo de Taylor

La diferenciación numérica es una herramienta fundamental en el análisis matemático que permite aproximar las derivadas de funciones cuando estas no pueden ser expresadas de manera exacta o cuando se trabaja con datos discretos. En este contexto, la diferenciación numérica juega un papel decisivo, ya que proporciona métodos que permiten obtener estimaciones de las tasas de cambio de funciones complejas.

Por otro lado, el desarrollo de Taylor es una técnica matemática que permite representar funciones analíticas como una suma infinita de términos

calculados a partir de las derivadas de la función en un punto específico. Esta representación no solo ofrece una forma de aproximar funciones, sino que también es un medio poderoso para entender su comportamiento cercano a un punto dado. El desarrollo de Taylor se convierte en un aliado esencial en la diferenciación numérica, ya que permite obtener derivadas aproximadas de funciones complejas a través de la evaluación de sus valores y derivadas en un punto determinado (Morales y Cordero, 2014).

La diferenciación numérica es una herramienta esencial en el análisis matemático, especialmente cuando se trata de obtener derivadas de funciones que no pueden diferenciarse de manera analítica. La diferenciación numérica se refiere a la aproximación del valor de la derivada de una función mediante el uso de valores discretos de la función en puntos específicos. A diferencia de la diferenciación analítica, que se basa en reglas matemáticas y fórmulas, la diferenciación numérica se basa en la evaluación de la función en puntos cercanos. Esto es especialmente útil para funciones complicadas, experimentales o que no tienen una forma explícita.

Existen diferentes métodos para calcular derivadas numéricas, siendo los más comunes el método de diferencias finitas, que se basa en la fórmula de la derivada como el límite del cociente de diferencias. En su forma más simple, la derivada de una función $f(x)$ en un punto x se puede aproximar como:

$$
f'(x) \approx \frac{f(x+h) - f(x)}{h}
$$

donde h es un pequeño incremento en x.

La diferenciación numérica tiene una gran relevancia en diversas disciplinas, desde la ingeniería hasta la física y la economía. En muchos casos, las funciones que se desean analizar pueden ser complejas o no estar definidas analíticamente, lo que hace que la diferenciación numérica se convierta en la única opción viable para obtener tasas de cambio y otros análisis relacionados.

En el caso de la simulación de sistemas físicos, las posiciones y velocidades de los objetos pueden ser representadas por funciones que son difíciles de derivar de manera exacta. La diferenciación numérica permite a los científicos e ingenieros calcular aceleraciones y otros parámetros críticos en sus modelos. Entre los métodos más utilizados en la diferenciación numérica se encuentran:

i. *Diferencias hacia adelante:* Este método utiliza el valor de la función en el punto actual y el siguiente. Se define como:

$$
f'(x) \approx \frac{f(x+h) - f(x)}{h}
$$

ii. *Diferencias hacia atrás:* Este enfoque utiliza el valor de la función en el punto actual y el anterior. Se expresa como:

$$
f'(x) \approx \frac{f(x) - f(x-h)}{h}
$$

iii. *Diferencias centradas:* Este método es más preciso y utiliza los valores de la función en ambos lados del punto de interés. Se formula así:

$$
f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}
$$

Cada uno de estos métodos tiene sus propias ventajas y desventajas en términos de precisión, estabilidad y requerimientos computacionales, lo que hace necesario seleccionar el método adecuado según las características de la función en estudio y el contexto de la aplicación. La diferenciación numérica es un componente esencial del análisis matemático aplicado, y su comprensión es relevante para la resolución de problemas en múltiples campos del conocimiento.

El desarrollo de Taylor es una herramienta fundamental en el análisis matemático que permite aproximar funciones mediante polinomios. Este método es particularmente útil no solo en el cálculo de valores de funciones en puntos cercanos a un valor conocido, sino también en la diferenciación numérica, donde se busca obtener derivadas aproximadas (Luna et al., 2024). El desarrollo de Taylor de una función \(f(x) \) que es suficientemente suave en torno a un punto \(a \) se delimita como la representación de \(f(x) \) como una serie infinita de términos que involucran las derivadas de \(f \) en el punto \(a \). Matemáticamente, se expresa como:

\[
f(x) = f(a) + f'(a)(x - a) + \frac{f''(a)}{2!}(x - a)^2 + \frac{f'''(a)}{3!}(x - a)^3 + \ldots
\]

Los términos de esta serie representan las distintas derivadas de la función en el punto \(a \), multiplicadas por potencias de \((x - a) \) y divididas por el factorial correspondiente. Esta representación permite aproximar \(f(x) \) mediante un polinomio de grado \(n \), donde el número de términos se puede ajustar según el nivel de precisión deseado. El desarrollo de Taylor presenta varias propiedades notables:

i. *Convergencia:* Bajo ciertas condiciones, la serie de Taylor converge a la función original en un intervalo alrededor del punto \(a \). Esta propiedad es crucial para su aplicación en la aproximación de funciones.

ii. *Exactitud:* Cuantos más términos se incluyan en el desarrollo, más precisa será la aproximación. Sin embargo, la convergencia y la exactitud dependen de la elección del punto \(a \) y de la naturaleza de la función.

iii. *Derivación:* El desarrollo de Taylor puede ser utilizado para calcular derivadas de funciones en puntos específicos, lo que es especialmente útil en contextos donde la derivada no se puede obtener de forma analítica.

El desarrollo de Taylor juega un papel trascendental en la diferenciación numérica al proporcionar una base teórica para estimar derivadas. Al utilizar la expansión de Taylor, se pueden derivar fórmulas de aproximación que permiten calcular la derivada de una función a partir de valores en puntos cercanos. Por ejemplo, al considerar la expansión de Taylor de primer orden, se obtiene la fórmula de la diferenciación hacia adelante:

\[
f'(x) \approx \frac{f(x + h) - f(x)}{h}
\]

donde \(h \) es un pequeño incremento. De manera similar, se pueden formular aproximaciones hacia atrás y centradas, lo que permite mayor flexibilidad y precisión en el cálculo de derivadas. Además, el desarrollo de Taylor se utiliza para analizar el error en las aproximaciones de diferenciación numérica, proporcionando una manera de estimar cuán lejos está la derivada aproximada de la derivada real, lo que es crucial para muchas aplicaciones en ingeniería y física. El desarrollo de Taylor no solo es una herramienta poderosa para la aproximación de funciones, sino que también se integra de manera efectiva con la diferenciación numérica, mejorando la precisión y versatilidad de los métodos numéricos.

La integración de la diferenciación numérica con el desarrollo de Taylor permite abordar problemas complejos en matemáticas y ciencias aplicadas, ofreciendo un enfoque robusto para estimar derivadas y resolver ecuaciones

diferenciales (Çevik et al., 2025). Esta sinergia no solo enriquece el análisis numérico, sino que también proporciona herramientas efectivas para la modelación y simulación en diversas disciplinas. Un ejemplo claro de la integración de estos conceptos se encuentra en la aproximación de funciones en puntos específicos. Cuando se desea calcular la derivada de una función en un punto x_0, se puede utilizar el desarrollo de Taylor para expresar la función en términos de su valor y sus derivadas en x_0. Si $f(x)$ es una función suficientemente suave, el desarrollo de Taylor hasta el primer orden se puede escribir como:

$$ f(x) \approx f(x_0) + f'(x_0)(x - x_0) $$

De esta forma, la derivada $f'(x_0)$ puede ser estimada utilizando valores de la función en puntos cercanos a x_0. En situaciones en las que no se dispone de una expresión analítica de la derivada, los métodos de diferenciación numérica, como las diferencias finitas, pueden complementarse con el desarrollo de Taylor para mejorar la precisión de las estimaciones. Otro ejemplo se encuentra en la resolución de ecuaciones diferenciales. Al aplicar métodos numéricos, como el método de Euler o el método de Runge-Kutta, se pueden utilizar desarrollos de Taylor para derivar las fórmulas de aproximación. Esto permite obtener resultados más precisos al considerar términos adicionales en el desarrollo que representan mejor el comportamiento de la solución en intervalos cortos.

La integración de la diferenciación numérica y el desarrollo de Taylor presenta varias ventajas. En primer lugar, mejora la precisión de las aproximaciones, ya que los términos adicionales en el desarrollo de Taylor pueden capturar mejor la curvatura de la función. En segundo lugar, proporciona un marco teórico sólido para entender el error en las estimaciones, lo que permite ajustar los métodos numéricos según las necesidades específicas de cada problema.

Sin embargo, este enfoque incluso tiene desventajas, la complejidad de los cálculos puede aumentar significativamente, especialmente si se consideran términos de orden superior en el desarrollo de Taylor. Además, la

elección de la cantidad de términos a incluir en el desarrollo puede ser crítica; un número insuficiente puede llevar a errores significativos, pese a que un número excesivo puede complicar los cálculos y requerir más recursos computacionales.

Las perspectivas futuras en la integración de la diferenciación numérica y el desarrollo de Taylor son prometedoras. La trascendencia de la computación y la inteligencia artificial, se están desarrollando algoritmos más sofisticados que combinan estas técnicas para abordar problemas cada vez más complejos. Además, la exploración de nuevas metodologías, como el uso de análisis de datos y técnicas de aprendizaje automático, puede revolucionar la forma en que se aplican estos conceptos en la práctica.

3.3 Aproximación por Diferencias: Implementaciones en GNU Octave y Python

La aproximación por diferencias es una técnica fundamental en el análisis numérico que permite estimar derivadas de funciones mediante el uso de valores discretos. En la actualidad los datos se generan y almacenan en formatos digitales, la capacidad de obtener derivadas de manera efectiva y precisa se vuelve decisivo para una amplia gama de aplicaciones, desde la ingeniería hasta la economía y las ciencias naturales. La aproximación por diferencias es una técnica que se utiliza para estimar derivadas de funciones a partir de valores discretos. Esta metodología es especialmente útil en situaciones donde no se dispone de una expresión analítica de la función o cuando se trabaja con datos experimentales.

La aproximación por diferencias se basa en la idea de que la derivada de una función en un punto puede ser estimada utilizando los valores de la función en puntos cercanos. Esto se logra mediante la formulación de diferencias finitas, que son expresiones que representan la variación de la función en términos de intervalos discretos. Existen varios tipos de aproximaciones por diferencias, entre las cuales las más comunes son:

- Diferencia hacia adelante: Esta aproximación utiliza el valor de la función en un punto y el valor en un punto posterior para estimar la derivada.

- Diferencia hacia atrás: A diferencia de la anterior, esta utiliza el valor de la función en un punto y el valor en un punto anterior.

- Diferencia central: Esta técnica combina los valores de la función en puntos anteriores y posteriores, proporcionando una estimación más precisa de la derivada.

Matemáticamente, la derivada de una función \(f(x) \) en un punto \(x \) puede aproximarse como:

- Diferencia hacia adelante:

\[f'(x) \approx \frac{f(x+h) - f(x)}{h} \]

- Diferencia hacia atrás:

\[f'(x) \approx \frac{f(x) - f(x-h)}{h} \]

- Diferencia central:

\[f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \]

donde \(h \) es un pequeño incremento en \(x \).

La aproximación por diferencias es relevante en el campo del análisis numérico porque permite resolver problemas que, de otro modo, serían inabordables mediante métodos analíticos. Se aplica en diversas disciplinas, como la física, la ingeniería y la economía, donde se requiere el análisis de cambios en sistemas complejos. Al permitir la estimación de derivadas y la resolución de ecuaciones diferenciales, esta técnica facilita la simulación y el modelado de fenómenos naturales y procesos industriales (González, 2008). Además, la aproximación por diferencias es fundamental en la discretización de problemas continuos, lo que permite el uso de computadoras para obtener soluciones numéricas. Esto es particularmente relevante en la modelización de sistemas donde los datos se obtienen de manera discreta, como en experimentos físicos o en el análisis de series temporales. Las aplicaciones de la aproximación por diferencias son vastas y variadas. Algunos ejemplos incluyen:

i. *Cálculo de velocidades y aceleraciones:* En física, se puede utilizar para estimar la velocidad de un objeto a partir de su posición en diferentes momentos.

ii. *Análisis de series temporales:* En economía y finanzas, se puede aplicar para identificar tendencias y cambios en datos históricos, como tasas de crecimiento o fluctuaciones en precios.

iii. *Simulación de modelos matemáticos:* En ingeniería, la técnica se utiliza para resolver ecuaciones diferenciales que describen sistemas dinámicos, como circuitos eléctricos o sistemas mecánicos.

iv. *Optimización de funciones:* En campos como la inteligencia artificial, la aproximación por diferencias se utiliza para calcular gradientes y optimizar funciones objetivo en algoritmos de aprendizaje automático.

La aproximación por diferencias se puede implementar utilizando una función que calcule la derivada de una función dada en un punto específico. La fórmula básica para la derivada aproximada utilizando diferencias finitas es:

- Diferencia hacia adelante:

$$
f'(x) \approx \frac{f(x + h) - f(x)}{h}
$$

- Diferencia hacia atrás:

$$
f'(x) \approx \frac{f(x) - f(x - h)}{h}
$$

- Diferencia centrada:

$$
f'(x) \approx \frac{f(x + h) - f(x - h)}{2h}
$$

Donde \(h \) es un pequeño número que representa el tamaño del paso. A continuación, se presenta una estructura básica del código en GNU Octave para implementar estas aproximaciones:

```
octave
function df = aproximacion_diferencias(func, x, h, metodo)
    switch metodo
        case 'adelante'
            df = (func(x + h) - func(x)) / h;
        case 'atras'
            df = (func(x) - func(x - h)) / h;
        case 'centrada'
            df = (func(x + h) - func(x - h)) / (2  h);
        otherwise
            error('Método no válido. Use "adelante", "atras" o "centrada".');
    end
end
```

Para ilustrar la implementación, consideremos la función \(f(x) = x^2 \) y calculemos la derivada en el punto \(x = 2 \) utilizando el método de aproximación centrada.

Primero, definamos la función:

```
octave
function y = f(x)
    y = x^2;
```

```
end
```

Luego, utilicemos la función de aproximación de diferencias:

```
octave
h = 0.01;  % Tamaño del paso
x = 2;     % Punto en el que queremos calcular la derivada

derivada_adelante = aproximacion_diferencias(@f, x, h, 'adelante');
derivada_atras = aproximacion_diferencias(@f, x, h, 'atras');
derivada_centrada = aproximacion_diferencias(@f, x, h, 'centrada');

fprintf('Derivada (adelante): %f\n', derivada_adelante);
fprintf('Derivada (atras): %f\n', derivada_atras);
fprintf('Derivada (centrada): %f\n', derivada_centrada);
```

Al ejecutar este código, obtendremos las aproximaciones de la derivada en el punto especificado. Este enfoque no solo demuestra la funcionalidad del algoritmo, sino que de igual modo ilustra cómo se puede aplicar en un entorno real de programación. Python se ha consolidado como uno de los lenguajes de programación más populares en el ámbito del análisis numérico y la ciencia de datos. Su sintaxis sencilla y las potentes bibliotecas disponibles facilitan la implementación de algoritmos complejos, como la aproximación por diferencias.

Para realizar la aproximación por diferencias en Python, es fundamental contar con algunas bibliotecas populares que facilitan los cálculos numéricos y la manipulación de datos. Las más relevantes son

NumPy y Matplotlib. Para instalarlas, es recomendable utilizar pip, el gestor de paquetes de Python. Ejecuta los siguientes comandos en tu terminal:

```
bash
pip install numpy matplotlib
```

Con estas librerías instaladas, estarás listo para realizar cálculos numéricos y visualizar los resultados. La implementación del algoritmo de aproximación por diferencias en Python sigue una estructura clara y concisa. En seguida, se presenta un esqueleto básico del código que puedes utilizar como referencia:

```
python
import numpy as np

def aproximacion_por_diferencias(funcion, x, h=1e-5):
    """
    Aproxima la derivada de una función en un punto dado utilizando diferencias finitas.

    Args:
        funcion: La función de la cual se desea calcular la derivada.
        x: El punto en el que se desea evaluar la derivada.
        h: El tamaño del paso para la aproximación.

    Returns:
        Derivada aproximada en el punto x.
```

```
    return (funcion(x + h) - funcion(x - h)) / (2  h)
```

En este código, definimos una función llamada aproximacion_por_diferencias que toma como argumentos otra función, un punto x y un tamaño de paso h. Utilizamos la fórmula de diferencias centradas para calcular la derivada en el punto especificado. Para ilustrar la implementación, consideremos la función \(f(x) = x^2 \) y calculemos su derivada en el punto \(x = 3 \). La derivada teórica de esta función es \(f'(x) = 2x \), que en \(x = 3 \) es igual a 6. Seguidamente, se muestra un ejemplo completo que incluye la visualización del resultado:

```
python

import numpy as np

import matplotlib.pyplot as plt


 Definimos la función

def funcion(x):

    return x2


 Calculamos la derivada aproximada

x = 3

derivada_aproximada = aproximacion_por_diferencias(funcion, x)


 Mostramos los resultados

print(f"La derivada aproximada de f(x) en x={x} es: {derivada_aproximada}")


 Visualización
```

```
x_vals = np.linspace(0, 6, 100)
y_vals = funcion(x_vals)

plt.plot(x_vals, y_vals, label='f(x) = x^2')
plt.scatter([x], [funcion(x)], color='red')  Punto de evaluación
plt.title('Gráfica de la función y su derivada')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.axhline(0, color='black', linewidth=0.5, ls='--')
plt.axvline(0, color='black', linewidth=0.5, ls='--')
plt.legend()
plt.grid()
plt.show()
```

En este ejemplo, se reduce la función \(f(x) = x^2 \), se calcula la derivada en \(x = 3 \) utilizando la función de aproximación por diferencias, y se visualiza la gráfica de la función junto con el punto evaluado. Al ejecutar este script, obtendrás tanto el valor de la derivada aproximada como una representación gráfica que ilustra la función y el punto de interés. Con esta implementación, has aprendido a utilizar Python para llevar a cabo la aproximación por diferencias, lo que te permitirá analizar y resolver problemas numéricos de manera eficiente y visualmente atractiva.

A través de la implementación práctica en GNU Octave, hemos visto cómo su sintaxis y estructura de código pueden ser utilizadas para calcular derivadas de manera efectiva. Por otro lado, Python, con sus potentes librerías y su versatilidad, ofrece una alternativa igualmente robusta para quienes prefieren trabajar en este entorno. La comparación de ambas implementaciones no solo resalta las similitudes en la lógica detrás del

algoritmo, sino que encima pone de manifiesto las peculiaridades y ventajas de cada lenguaje.

3.4 Derivadas Parciales por Diferencias: Implementación y Comparativa en GNU Octave y Python

La aproximación de derivadas parciales es un tema fundamental en el análisis numérico, ya que permite estimar la variación de funciones multivariables respecto a varias variables independientes. Las derivadas parciales son una extensión del concepto de derivadas a funciones que dependen de más de una variable (Thomas, 2010). Para una función $f(x, y)$, la derivada parcial respecto a x se define como el límite de la razón de cambio de f cuando se varía x en tanto se mantiene y constante. Matemáticamente, se expresa como:

$$
\frac{\partial f}{\partial x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x, y) - f(x, y)}{\Delta x}
$$

De manera similar, podemos definir la derivada parcial respecto a y. Este concepto es esencial, ya que permite analizar cómo la función se comporta en función de cada variable por separado, lo que resulta útil en la optimización y en el análisis de sistemas complejos. En el contexto del análisis numérico, las derivadas parciales son esenciales para la resolución de ecuaciones diferenciales parciales y para la implementación de algoritmos de optimización que requieren gradientes. Sin embargo, el cálculo analítico de derivadas parciales puede volverse impráctico para funciones complejas o para aquellos modelos donde la forma explícita de la función no es conocida.

Por esta razón, las técnicas de aproximación, como las diferencias finitas, se convierten en herramientas valiosas que permiten a los investigadores y profesionales obtener estimaciones de derivadas de manera computacional. La aproximación de derivadas parciales mediante diferencias

finitas es una técnica fundamental en el análisis numérico y se utiliza ampliamente en diversas aplicaciones científicas e ingenierías.

El método de diferencias hacia adelante es una técnica sencilla y directa para aproximar la derivada parcial de una función. Este método utiliza el valor de la función en un punto y en un punto adyacente. La fórmula para la derivada parcial de una función $f(x, y)$ respecto a x se expresa como:

$$
\frac{\partial f}{\partial x} \approx \frac{f(x + h, y) - f(x, y)}{h}
$$

donde h es un pequeño incremento en la variable x. Este método es particularmente útil cuando se necesita evaluar la derivada en un punto específico y se tiene acceso a los valores de la función en puntos cercanos. El método de diferencias hacia atrás es otra técnica de aproximación que se basa en el valor de la función en el punto actual y en un punto anterior. La fórmula para la derivada parcial de $f(x, y)$ respecto a x se puede escribir como:

$$
\frac{\partial f}{\partial x} \approx \frac{f(x, y) - f(x - h, y)}{h}
$$

Al igual que con el método de diferencias hacia adelante, h debe ser un valor pequeño. Este enfoque es útil cuando se desea evaluar la derivada en un punto, pero se tiene más información sobre el comportamiento de la función en puntos previos. El método de diferencias centradas combina las ideas de las diferencias hacia adelante y hacia atrás, proporcionando una aproximación más precisa de la derivada. La fórmula para la derivada parcial de $f(x, y)$ respecto a x se reduce como:

\[

\frac{\partial f}{\partial x} \approx \frac{f(x + h, y) - f(x - h, y)}{2h}

\]

Este método utiliza los valores de la función en ambos lados del punto en cuestión, lo que permite cancelar errores de aproximación que pueden estar presentes en los métodos de diferencias hacia adelante y hacia atrás. Es especialmente ventajoso en situaciones donde se busca una mayor precisión en la estimación de la derivad. La implementación de métodos de aproximación por diferencias finitas en GNU Octave es un proceso relativamente sencillo gracias a su sintaxis intuitiva y a su naturaleza similar a MATLAB. ¿Cómo implementar la aproximación de derivadas parciales utilizando el método de diferencias centradas? supongamos que deseamos calcular la derivada parcial de una función \(f(x, y) = x^2 + y^2 \) con respecto a \(x \) en un punto específico.

octave

```
pkg load symbolic
syms x y;  % Define x e y como variables simbólicas
f = x^2 + y^2; % Define la función
df_dx = diff(f, x); % Calcula la derivada parcial con respecto a x
df_dy = diff(f, y); % Calcula la derivada parcial con respecto a y


disp(df_dx);
disp(df_dy);
```

Este código define una función derivada_parcial_x que toma como argumentos la función \(f \), las coordenadas \(x \) y \(y \), y el tamaño del paso \(h \). Utiliza la fórmula de diferencias centradas para calcular la derivada parcial y la imprime en la consola. Para visualizar los resultados de nuestras aproximaciones, podemos utilizar la función de graficado de GNU Octave. En el caso de, si queremos graficar la función \(f(x, y) \) en un rango dado y observar cómo se comporta la derivada en diferentes puntos, podemos proceder de la siguiente manera:

octave

```
% Definir los vectores de coordenadas
x = -5:0.5:5;
y = -3:0.2:3;

% Crear la malla de puntos
[X, Y] = meshgrid(x, y);

% Visualizar la malla (opcional)
figure; % Crea una nueva ventana de figura
plot(X, Y, 'r.'); % Gráfico de puntos rojos
title('Malla de puntos en Octave');
xlabel('X');
ylabel('Y');
grid on;
```

Este fragmento de código crea una superficie tridimensional de la función y resalta el punto donde se calculó la derivada parcial. La visualización es una parte trascendental en el análisis numérico, ya que

permite entender mejor el comportamiento de la función y sus derivadas en diferentes regiones del espacio. Con estas herramientas y ejemplos, los usuarios de GNU Octave podrán implementar y visualizar la aproximación de derivadas parciales por diferencias finitas de manera efectiva.

Para llevar a cabo la implementación de la aproximación de derivadas parciales por diferencias finitas en Python, es esencial contar con un entorno adecuado. Python es un lenguaje muy popular en la comunidad científica y de análisis numérico debido a su simplicidad y a la vasta cantidad de bibliotecas disponibles. En este caso, vamos a utilizar las bibliotecas NumPy y Matplotlib, que son fundamentales para realizar cálculos numéricos y visualización de datos, respectivamente. Para instalar estas bibliotecas, se pueden utilizar los siguientes comandos en la terminal:

```
bash
pip install numpy matplotlib
```

En seguida, se presenta un ejemplo de código que implementa la aproximación de la derivada parcial de una función \(f(x, y) \) utilizando el método de diferencias centradas. Consideremos la función \(f(x, y) = x^2 + y^2 \). Para calcular la derivada parcial con respecto a \(x \) y \(y \), el código sería el siguiente:

```
python
def f(x, y):
    return x**2 + y**2  # Return the sum of squares of x and y

def derivada_parcial_x(f, x, y, h=1e-5):
    return (f(x + h, y) - f(x - h, y)) / (2 * h)  # Use * for multiplication
```

```
def derivada_parcial_y(f, x, y, h=1e-5):
    return (f(x, y + h) - f(x, y - h)) / (2 * h)  # Use * for multiplication

# Points where we will calculate the partial derivatives
x = 1.0
y = 2.0

# Calculate the partial derivatives
df_dx = derivada_parcial_x(f, x, y)
df_dy = derivada_parcial_y(f, x, y)

print(f"Derivada parcial de f con respecto a x en ({x}, {y}): {df_dx}")
print(f"Derivada parcial de f con respecto a y en ({x}, {y}): {df_dy}")
```

Este código define la función \(f \) y dos funciones que calculan las derivadas parciales con respecto a \(x \) y \(y \) utilizando el método de diferencias centradas. Luego, se calcula y se imprime el resultado de las derivadas parciales en un punto específico. Para visualizar los resultados, podemos utilizar Matplotlib para graficar la función y sus derivadas parciales. Después, se presenta un código adicional que muestra cómo generar un gráfico de la función \(f(x, y) \) y cómo las derivadas parciales afectan la pendiente en el plano:

```
python
# Define the function f(x, y)
def f(x, y):
    return x**2 + y**2  # Example function: f(x, y) = x^2 + y^2
```

```
# Visualize the function in a text-based grid
def visualize():
    for y in range(5, -6, -1):  # Start from 5 to -5 for y-axis
        line = ""  # Reset the line for each row
        for x in range(-5, 6):  # From -5 to 5 for x-axis
            z = f(x, y) / 10  # Scale down z-axis values for visibility
            if (x, y) == (1, 2):  # Highlight point (1, 2)
                line += " R "  # "R" for red point of interest
            else:
                line += " . "  # "." for regular points
        print(line)  # Print the line for the current y value

# Call the visualize function
visualize()
```

Este script crea un gráfico tridimensional de la función, mostrando la superficie generada por \(f(x, y) \) y destacando el punto donde se calcularon las derivadas parciales. La visualización proporciona una comprensión intuitiva de cómo la función se comporta en un entorno tridimensional, así como la relación entre la función y sus derivadas. La implementación en Python no solo es directa y accesible, sino que también permite realizar cálculos complejos de manera eficiente, lo que la convierte en una herramienta poderosa para el análisis numérico y la simulación.

Capítulo IV

Integración numérica con Python

La integración numérica, fundamental para las matemáticas aplicadas, se ocupa de la aproximación de integrales definidas y no definidas utilizando métodos computacionales. En lugar de calcular el valor exacto de una integral, que a menudo puede ser complicado o incluso imposible de obtener analíticamente, los métodos numéricos permiten obtener estimaciones precisas que se pueden calcular con un ordenador. Esta técnica es esencial en diversos campos, como la ingeniería, la física, la economía y la biología, donde los modelos matemáticos a menudo requieren la integración de funciones complejas.

La importancia de la integración numérica en la computación científica radica en su capacidad para manejar problemas intrínsecamente difíciles. Muchas ecuaciones diferenciales, así, no pueden resolverse de forma analítica y requieren el uso de la integración numérica para obtener soluciones aproximadas. Asimismo, en la simulación de fenómenos naturales y en la optimización de sistemas, la integración numérica se convierte en una herramienta vital, ya que permite la evaluación de áreas bajo curvas, volúmenes de sólidos y otros conceptos que son esenciales para el análisis cuantitativo.

4.1 Métodos de integración numérica

La integración numérica se refiere a una variedad de técnicas utilizadas para aproximar el valor de una integral, especialmente cuando no es posible obtener una solución analítica. Existen múltiples métodos, cada uno con sus ventajas y desventajas, que se adaptan a diferentes tipos de funciones y contextos. El método del trapecio es uno de los más simples y ampliamente utilizados para la integración numérica. Este método se basa en la idea de aproximar la región bajo la curva de una función por un trapezoide, en lugar de utilizar rectángulos como en la regla de los rectángulos (Casteleiro, 2002). Al dividir el intervalo de integración en n subintervalos, se calcula el área de cada trapezoide y se suman estas áreas para obtener una

aproximación del valor total de la integral. La fórmula del método del trapecio se puede expresar como:

\[

\int_a^b f(x) \, dx \approx \frac{b - a}{2n} \left[f(a) + 2 \sum_{i=1}^{n-1} f(x_i) + f(b) \right]

\]

donde \(x_i = a + i \cdot \Delta x \) y \(\Delta x = \frac{b - a}{n} \).

Para implementar el método del trapecio en Python, se puede utilizar una función sencilla que reciba como parámetros la función a integrar, los límites de integración y el número de subintervalos. Para nuestra:

```
python
import numpy as np

def metodo_trapecio(f, a, b, n):
    x = np.linspace(a, b, n + 1)
    y = f(x)
    integral = (b - a) / (2  n)  (y[0] + 2  np.sum(y[1:-1]) + y[-1])
    return integral

 Ejemplo de uso
f = lambda x: x2
resultado = metodo_trapecio(f, 0, 1, 100)
print(f'Integral aproximada: {resultado}')
```

Supongamos que queremos calcular la integral de \(f(x) = x^2 \) en el intervalo [0, 1]. Utilizando el método del trapecio con 100 subintervalos, obtendremos una aproximación de la integral. El resultado debería aproximarse a \(\frac{1}{3}\) o aproximadamente 0.3333.

La regla de Simpson es otro método popular para la integración numérica que ofrece una mayor precisión que el método del trapecio, especialmente para funciones suaves. Este método usa polinomios de segundo grado para aproximar la función en cada par de intervalos y se basa en la idea de interpolar los puntos mediante parábolas (Ezquerro, 2012). La fórmula de la regla de Simpson se expresa como:

\[
\int_a^b f(x) \, dx \approx \frac{b - a}{6n} \left[f(a) + 4 \sum_{i=1}^{n} f(x_{2i-1}) + 2 \sum_{i=1}^{n-1} f(x_{2i}) + f(b) \right]
\]

donde \(n \) debe ser par y \(x_i = a + i \cdot \Delta x \).

La implementación de la regla de Simpson en Python es similar a la del método del trapecio, pero requiere un ajuste para manejar los coeficientes de la suma. En seguida, se presenta un ejemplo:

```python
def regla_simpson(f, a, b, n):
    if n % 2 == 1:
        raise ValueError("n debe ser par")
    x = np.linspace(a, b, n + 1)
    y = f(x)
```

```
    integral = (b - a) / (6  n)  (y[0] + 4  np.sum(y[1:-1:2]) + 2  np.sum(y[2:-2:2]) +
y[-1])
    return integral

Ejemplo de uso
resultado_simpson = regla_simpson(f, 0, 1, 100)
print(f'Integral aproximada por Simpson: {resultado_simpson}')
```

Siguiendo el mismo ejemplo de \(f(x) = x^2 \) en el intervalo [0, 1] y utilizando 100 subintervalos, la regla de Simpson proporcionará una aproximación más precisa de la integral.

Los métodos adaptativos son técnicas de integración que ajustan dinámicamente el tamaño de los subintervalos según la variabilidad de la función a integrar. En regiones donde la función presenta cambios bruscos, se usan subintervalos más pequeños, en tanto que en regiones suaves se pueden utilizar subintervalos más grandes (Canchoa, 2009). La implementación de un método adaptativo puede ser más compleja, pero un enfoque común es usar la regla del trapecio o Simpson de forma recursiva, ajustando los subintervalos según el error estimado. Aquí tienes un ejemplo básico utilizando el método del trapecio:

```
python
def metodo_adaptativo(f, a, b, tol, max_iter):
    def trapecio_recursivo(a, b, tol):
        n = 1
        integral = metodo_trapecio(f, a, b, n)
        while True:
            n = 2
```

```
        nuevo_integral = metodo_trapecio(f, a, b, n)
        if np.abs(nuevo_integral - integral) < tol or n > max_iter:
            break
        integral = nuevo_integral
    return integral

return trapecio_recursivo(a, b, tol)
```

Ejemplo de uso

```
resultado_adaptativo = metodo_adaptativo(f, 0, 1, 1e-5, 10)
print(f'Integral aproximada adaptativa: {resultado_adaptativo}')
```

Utilizando el método adaptativo para calcular la integral de $f(x) = x^2$ en el intervalo [0, 1] con una tolerancia de 1×10^{-5}, el resultado debería ser muy cercano a $\frac{1}{3}$. Estos métodos de integración numérica proporcionan herramientas versátiles y efectivas para abordar problemas de integración en diversas aplicaciones científicas y de ingeniería. La elección del método adecuado depende de la función a integrar y los requisitos de precisión y eficiencia.

NumPy es una biblioteca fundamental para la computación numérica en Python. Aunque no está diseñada específicamente para la integración, proporciona funciones que son útiles para realizar cálculos matemáticos que a menudo se relacionan con la integración. En efecto, la función numpy.trapz permite calcular la integral definida de un conjunto de puntos usando el método del trapecio, y numpy.cumsum puede ser útil para sumas acumuladas que contribuyen a la integración. Entonces, se muestra un ejemplo de cómo utilizar NumPy para calcular la integral de una función simple, como $f(x) = x^2$, en el intervalo de 0 a 1:

```
python
import numpy as np

 Definimos el rango y el número de puntos
x = np.linspace(0, 1, 100)
y = x2

 Calculamos la integral usando el método del trapecio
integral = np.trapz(y, x)
print(f"La integral de x^2 desde 0 hasta 1 es aproximadamente: {integral}")
```

SciPy es una biblioteca que se basa en NumPy y proporciona un conjunto más amplio de herramientas científicas, incluyendo funciones específicas para la integración. El submódulo scipy.integrate incluye métodos como quad para integración adaptativa, dblquad para integrales dobles y odeint para resolver ecuaciones diferenciales. Aquí hay un ejemplo de cómo utilizar scipy.integrate.quad para calcular la integral de \(f(x) = \sin(x) \) en el intervalo de 0 a \(\pi \):

```
python
from scipy.integrate import quad
import numpy as np

 Definimos la función a integrar
def f(x):
    return np.sin(x)
```

```
Calculamos la integral
resultado, error = quad(f, 0, np.pi)
print(f"La integral de sin(x) desde 0 hasta pi es aproximadamente: {resultado}")
```

Si bien NumPy ofrece funciones básicas para la integración, SciPy proporciona métodos más sofisticados y precisos. En spi, scipy.integrate.quad está diseñado para manejar integrales complicadas y puede ofrecer estimaciones de error, lo que no está disponible en las funciones de NumPy. SymPy es una biblioteca de Python para matemáticas simbólicas, lo que significa que puede realizar cálculos algebraicos con variables en lugar de solo números. Esto incluye la capacidad de realizar integración simbólica, lo que es útil cuando se desea obtener una expresión analítica de una integral. He aquí un modelo de cómo usar SymPy para calcular la integral simbólica de \(f(x) = x^2 \):

```
python
from sympy import symbols, integrate

Definimos la variable simbólica
x = symbols('x')

Definimos la función
funcion = x2

Calculamos la integral
integral = integrate(funcion, (x, 0, 1))
print(f"La integral simbólica de x^2 desde 0 hasta 1 es: {integral}")
```

SymPy es especialmente útil en campos donde se requiere una comprensión profunda de la integral, como en la física teórica o la ingeniería. Permite a los investigadores obtener soluciones exactas y trabajar con derivadas e integrales en un formato legible. NumPy es ideal para cálculos básicos y operaciones con matrices, SciPy proporciona métodos avanzados y precisos, y SymPy se destaca en la integración simbólica. La elección de la biblioteca dependerá de las necesidades específicas del usuario y de la complejidad de los problemas que se deseen resolver.

4.2 Integración de Newton-Cotes y Cuadraturas de Gauss con GNU Octave y Python

La integración numérica es una herramienta fundamental en diversas ramas de la ciencia y la ingeniería, ya que permite calcular áreas, volúmenes y otras magnitudes que no pueden ser evaluadas de forma analítica. Dentro de este campo, las fórmulas de Newton-Cotes y las cuadraturas de Gauss son dos de los métodos más utilizados para aproximar integrales definidas. Las fórmulas de Newton-Cotes son un conjunto de métodos de integración numérica que se basan en la interpolación polinómica. Estas fórmulas se derivan de la idea de aproximar el integrando mediante un polinomio que pasa a través de un conjunto de puntos conocidos. El nombre "Newton-Cotes" proviene de los matemáticos Isaac Newton y Richard Cotes, quienes contribuyeron al desarrollo de estas técnicas en el siglo XVIII.

Existen dos categorías principales de fórmulas de Newton-Cotes: las fórmulas abiertas y las fórmulas cerradas. Las fórmulas cerradas utilizan los extremos del intervalo de integración como puntos de evaluación, pese a que las fórmulas abiertas utilizan puntos que no incluyen los extremos del intervalo. Estas fórmulas se pueden clasificar según el número de puntos utilizados para la interpolación, lo que resulta en diferentes órdenes de precisión.

Las cuadraturas de Gauss son un método más sofisticado para la aproximación de integrales que se basa en la teoría de los polinomios ortogonales. En lugar de utilizar puntos equidistantes como en las fórmulas de Newton-Cotes, las cuadraturas de Gauss seleccionan puntos de evaluación

(llamados nodos) y pesos asociados de manera que maximicen la precisión de la aproximación. Este enfoque permite que las cuadraturas de Gauss sean especialmente efectivas para funciones que son suaves y continúas, ofreciendo una precisión superior en comparación con las fórmulas de Newton-Cotes del mismo grado.

El desarrollo de las cuadraturas de Gauss se atribuye a Carl Friedrich Gauss, quien formuló estas técnicas en el siglo XIX. La más conocida de estas cuadraturas es la cuadratura de Gauss-Legendre, que se utiliza ampliamente debido a su capacidad para aproximar integrales definidas con un alto grado de precisión.

Las fórmulas de Newton-Cotes y las cuadraturas de Gauss son fundamentales en la integración numérica, ya que permiten a los investigadores y profesionales obtener resultados precisos y eficientes en una amplia gama de aplicaciones (Raffo et al., 2007). Desde la resolución de problemas en física e ingeniería hasta la modelización de fenómenos en biología y economía, estas técnicas son esenciales cuando se requiere evaluar integrales que no pueden ser calculadas de forma analítica.

La elección entre las fórmulas de Newton-Cotes y las cuadraturas de Gauss depende de varios factores, incluyendo la naturaleza de la función a integrar, el intervalo de integración y el nivel de precisión requerido. Ambas técnicas tienen sus ventajas y desventajas, y su correcta implementación puede marcar la diferencia en la calidad de los resultados obtenidos.

La implementación de las fórmulas de Newton-Cotes en GNU Octave es un proceso accesible que permite a los usuarios llevar a cabo integraciones numéricas de manera efectiva. Este software, que es una alternativa de código abierto a MATLAB, proporciona un entorno de programación amigable que facilita la experimentación con diferentes métodos de integración. Las fórmulas de Newton-Cotes se pueden clasificar en dos grupos: fórmulas cerradas y fórmulas abiertas.

Las fórmulas cerradas de Newton-Cotes, como el método del trapecio y la regla de Simpson, son algunas de las más utilizadas. Aquí se muestra cómo implementar la regla de Simpson en GNU Octave:

```
octave
function I = simpson(f, a, b, n)
    % f: función a integrar
    % a: límite inferior
    % b: límite superior
    % n: número de subintervalos (debe ser par)

    h = (b - a) / n; % ancho del subintervalo
    x = a:h:b; % puntos de evaluación
    y = f(x); % valores de la función en los puntos

    I = (h/3)  (y(1) + 4  sum(y(2:2:end-1)) + 2  sum(y(3:2:end-2)) + y(end));
end

% Ejemplo de uso
f = @(x) x.^2; % función a integrar
a = 0; % límite inferior
b = 1; % límite superior
n = 10; % número de subintervalos
resultado = simpson(f, a, b, n);
disp(['Resultado de la integral: ', num2str(resultado)]);
```

Las fórmulas abiertas, como la regla de Boole, son menos comunes, pero también se pueden implementar en GNU Octave. Aquí hay un ejemplo básico de la regla de Boole:

```
octave
function I = boole(f, a, b, n)
    % f: función a integrar
    % a: límite inferior
    % b: límite superior
    % n: número de subintervalos (debe ser múltiplo de 4)

    h = (b - a) / n; % ancho del subintervalo
    x = a:h:b; % puntos de evaluación
    y = f(x); % valores de la función en los puntos

    I = (2h/45) (7y(1) + 32sum(y(2:4:end-1)) + 12sum(y(3:4:end-2)) + 32sum(y(4:4:end-1)) + 7y(end));
end

% Ejemplo de uso
f = @(x) sin(x); % función a integrar
a = 0; % límite inferior
b = pi; % límite superior
n = 8; % número de subintervalos
resultado = boole(f, a, b, n);
disp(['Resultado de la integral: ', num2str(resultado)]);
```

Para evaluar la efectividad de las fórmulas de Newton-Cotes implementadas, es útil comparar los resultados obtenidos con diferentes métodos. Esto se puede hacer calculando la integral de una función conocida,

como \(f(x) = e^{-x^2} \), y comparando las aproximaciones generadas por la regla del trapecio, la regla de Simpson y la regla de Boole.

```
octave
f = @(x) exp(-x.^2); % función a integrar
a = 0; % límite inferior
b = 1; % límite superior
n = 10; % número de subintervalos

trapecio_resultado = trapecio(f, a, b, n);
simpson_resultado = simpson(f, a, b, n);
boole_resultado = boole(f, a, b, n);

disp(['Resultado con la regla del trapecio: ', num2str(trapecio_resultado)]);
disp(['Resultado con la regla de Simpson: ', num2str(simpson_resultado)]);
disp(['Resultado con la regla de Boole: ', num2str(boole_resultado)]);
```

Para implementar las cuadraturas de Gauss en Python, es crucial contar con las bibliotecas adecuadas que faciliten los cálculos numéricos, las bibliotecas más recomendadas son NumPy y SciPy, que proporcionan potentes herramientas para la manipulación de arreglos y la integración numérica. Para instalar estas bibliotecas, se puede utilizar el gestor de paquetes pip. He aquí el comando que se debe ejecutar en la terminal:

```
bash
pip install numpy scipy
```

Una vez instaladas, se pueden importar fácilmente en el script de Python:

```
python
import numpy as np
from scipy import integrate
```

La cuadratura de Gauss permite aproximar integrales definidas mediante la evaluación de la función en puntos estratégicos, conocidos como puntos de Gauss, y ponderando estas evaluaciones con los pesos correspondientes. Por loque la cuadratura de Gauss para integrar una función sencilla, como \(f(x) = e^{-x^2}\) en el intervalo \([-1, 1]\), se puede ver aquí:

```
python
import numpy as np
from scipy import integrate

 Definimos la función a integrar
def f(x):
    return np.exp(-x2)

 Realizamos la integración utilizando la cuadratura de Gauss
resultado, error = integrate.quad(f, -1, 1)

 Mostramos el resultado
print(f"Resultado de la integración: {resultado}")
```

```
print(f"Error estimado: {error}")
```

Este código utiliza la función quad de SciPy, que aplica la cuadratura de Gauss de manera automática. Al ejecutar este script, se obtiene un resultado que se aproxima al valor exacto de la integral. Para ilustrar cómo se pueden realizar integraciones utilizando diferentes órdenes de cuadratura de Gauss, se puede utilizar la función gauss_legendre de la biblioteca NumPy, que permite calcular los puntos y pesos de la cuadratura. Aquí hay un ejemplo que muestra cómo se puede implementar:

```
python

import numpy as np


 Número de puntos de Gauss

n = 3


 Obtenemos los puntos y pesos

puntos, pesos = np.polynomial.legendre.leggauss(n)


 Cambiamos el intervalo de integración

a = -1

b = 1

puntos_normalizados = 0.5  (b - a)  puntos + 0.5  (b + a)


 Calculamos la integral

integral = 0.5  (b - a)  np.sum(pesos  f(puntos_normalizados))
```

```
Mostramos el resultado
print(f"Resultado de la integración con cuadratura de Gauss de orden {n}: {integral}")
```

La precisión de la cuadratura de Gauss depende del número de puntos utilizados; generalmente, a mayor número de puntos, mayor será la precisión. Para evaluar la eficiencia, se pueden comparar los resultados obtenidos con diferentes órdenes de cuadratura y con el método de integración definido de SciPy (Linge y Langtangen, 2016). Es recomendable realizar un análisis de convergencia, donde se mida el error de la aproximación en función del número de puntos de Gauss utilizados. Esto se puede hacer calculando la integral para diferentes valores de \(n \) y observando cómo se comporta el error en cada caso. Las cuadraturas de Gauss son una herramienta poderosa en Python para la integración numérica, ofreciendo resultados precisos y eficientes, especialmente en funciones que son difíciles de integrar analíticamente.

4.3 Método de Integración de Simpson

El método de integración de Simpson es una técnica fundamental en el campo del cálculo numérico, utilizada para aproximar el valor de integrales definidas. Su desarrollo se remonta al siglo XVIII y ha sido una herramienta decisivo para matemáticos e ingenieros que requieren soluciones rápidas y eficaces para problemas que no pueden resolverse analíticamente.

El método lleva el nombre de Thomas Simpson, un matemático inglés que popularizó esta técnica en su obra "The Doctrine of Fluxions", publicada en 1750. Sin embargo, las raíces del método se pueden rastrear aún más atrás, en las contribuciones de matemáticos como Arquímedes, quien ya en la antigüedad utilizaba técnicas de aproximación para el cálculo de áreas. Simpson introdujo una forma más sistemática de interpolación polinómica, que se basa en la idea de dividir el intervalo de integración en segmentos más pequeños y aplicar un polinomio de segundo grado para estimar el área bajo la curva.

La relevancia del método de Simpson radica en su capacidad para proporcionar aproximaciones precisas con un número reducido de evaluaciones de la función. A diferencia de otros métodos de integración, como el método del trapecio, que utiliza rectángulos para aproximar el área, el método de Simpson emplea parábolas, lo que le permite seguir más de cerca la forma de la función (Abramowitz y Stegun, 1972). Esto generalmente resulta en un menor error de aproximación, especialmente para funciones suaves y continuas.

El método de Simpson se clasifica en dos variantes: el método de Simpson 1/3, que utiliza tres puntos para cada intervalo, y el método de Simpson 3/8, que utiliza cuatro puntos. Cada uno tiene sus contextos de uso específicos, pero ambos son ampliamente utilizados en aplicaciones donde la precisión es esencial, como en la ingeniería, la física y la economía.

El método de integración de Simpson encuentra aplicaciones en diversas disciplinas. En la ingeniería, se utiliza para calcular áreas y volúmenes en diseños estructurales, mientras que en la física se aplica en el análisis de trayectorias y en la resolución de problemas de dinámica de fluidos. En economía, se utiliza para estimar el valor presente de flujos de efectivo futuros, donde la integración es necesaria para evaluar el rendimiento de inversiones. A continuación, se presenta una implementación básica de este método en GNU Octave:

```
octave
function I = simpson(f, a, b, n)
    % f: función a integrar
    % a: límite inferior
    % b: límite superior
    % n: número de subintervalos (debe ser par)

    if mod(n, 2) ~= 0
```

```
    error('El número de subintervalos n debe ser par.');
  end

  h = (b - a) / n; % ancho de los subintervalos
  x = a:h:b;       % puntos de evaluación
  y = f(x);        % evaluación de la función en los puntos

  % Aplicación de la regla de Simpson
  I = (h/3)  (y(1) + 4sum(y(2:2:end-1)) + 2sum(y(3:2:end-2)) + y(end));
end
```

En este código, f es la función a integrar, a y b son los límites inferior y superior de la integral, y n es el número de subintervalos que se debe especificar como un número par. La función devuelve el valor aproximado de la integral. Para ilustrar la aplicación del método de Simpson en GNU Octave, consideremos la función \(f(x) = x^2 \) y calculemos la integral definida de esta función en el intervalo \([0, 1]\). Primero, definimos la función y luego llamamos a la función simpson que hemos creado:

```
octave
% Definición de la función
f = @(x) x.^2;

% Límite inferior y superior
a = 0;
b = 1;
```

```
% Número de subintervalos
n = 10;

% Cálculo de la integral usando el método de Simpson
resultado = simpson(f, a, b, n);
fprintf('El valor aproximado de la integral es: %.4f\n', resultado);
```

Al ejecutar este código, obtendremos un resultado que se aproxima al valor exacto de la integral \(\frac{1}{3}\), que es aproximadamente 0.3333. Este ejemplo demuestra la facilidad y eficacia del método de integración de Simpson en GNU Octave, permitiendo realizar cálculos numéricos de manera eficiente. El método de Simpson se basa en la aproximación de la integral de una función mediante un polinomio de segundo grado. La implementación en Python se puede realizar de la siguiente manera:

```
python
import numpy as np

def simpson_rule(f, a, b, n):
    if n % 2 == 1:  n debe ser par
        raise ValueError("El número de subintervalos 'n' debe ser par.")

    h = (b - a) / n  ancho de los subintervalos
    x = np.linspace(a, b, n + 1)  puntos de evaluación
    fx = f(x)  evaluación de la función en los puntos

     Aplicación de la regla de Simpson
```

```
    integral = h/3  (fx[0] + 4  np.sum(fx[1:n:2]) + 2  np.sum(fx[2:n-1:2]) + fx[n])

    return integral
```

En este código, la función simpson_rule toma como parámetros la función f a integrar, los límites de integración a y b, y el número de subintervalos n. Se asegura de que n sea par y calcula la integral usando la regla de Simpson. Para ilustrar cómo utilizar la implementación del método de Simpson en un caso práctico, consideremos la función \(f(x) = x^2 \) en el intervalo \([0, 1]\). Aquí se muestra un ejemplo de cómo aplicar nuestra función simpson_rule:

```
python

 Definimos la función a integrar

def f(x):

    return x2


 Límites de integración y número de subintervalos

a = 0

b = 1

n = 10   Debe ser par


 Cálculo de la integral

resultado = simpson_rule(f, a, b, n)


 Mostrar el resultado

print(f"La integral de f(x) desde {a} hasta {b} es aproximadamente: {resultado}")
```

Al ejecutar este código, se calculará y mostrará la integral de \(f(x) = x^2 \) en el intervalo \([0, 1]\) usando el método de Simpson, proporcionando un resultado que se aproximará a \(\frac{1}{3}\) o aproximadamente 0.3333.

- **EDO´s y la función ode45**

Resolver un conjunto de ecuaciones diferenciales ordinarias no rígidas (EDO no rígidas) con el conocido método explícito de Dormand-Prince de orden 4.

fun es un manejador de función, una función en línea o una cadena que contiene el nombre de la función que define la EDO: y' = f(t,y). La función debe aceptar dos entradas, donde la primera es el tiempo t y la segunda es un vector columna de incógnitas y. "trange" especifica el intervalo de tiempo en el que se evaluará la EDO, Normalmente, es un vector de dos elementos que especifica los tiempos inicial y final ([tinit, tfinal]). Si hay más de dos elementos, la solución también se evaluará en estos tiempos intermedios.

De forma predeterminada, ode45el integrate_adaptivealgoritmo utiliza un paso de tiempo adaptativo. La tolerancia para el cálculo del paso de tiempo se puede modificar mediante las opciones "RelTol" y "AbsTol". init contiene el valor inicial de las incógnitas. Si es un vector fila, la solución y será una matriz en la que cada columna es la solución del valor inicial correspondiente en init. El cuarto argumento opcional ode_opt especifica opciones no predeterminadas para el solucionador de EDO. Es una estructura generada por odeset.

La variable t es un vector columna que contiene los momentos en que se encontró la solución. La salida y es una matriz donde cada columna se refiere a una incógnita diferente del problema y cada fila corresponde a un momento en t. La salida también puede devolverse como una solución de estructura , que tiene un campo x que contiene un vector de filas de los momentos en que se evaluó la solución, y un campo y que contiene la matriz de la solución, de modo que cada columna corresponde a un momento en x.

Si se usa la "Events" opción, se pueden devolver tres salidas adicionales. te contiene el momento en que una función de evento devolvió un cero. ye contiene el valor de la solución en el momento te.ie contiene un índice que indica qué función de evento se activó en el caso de múltiples funciones de evento.

```
## Demonstrate convergence order for ode45
 tol = 1e-5 ./ 10.^[0:8];
 for i = 1 : numel (tol)
   opt = odeset ("RelTol", tol(i), "AbsTol", realmin);
   [t, y] = ode45 (@(t, y) -y, [0, 1], 1, opt);
   h(i) = 1 / (numel (t) - 1);
   err(i) = norm (y .* exp (t) - 1, Inf);
 endfor


 ## Estimate order visually
 loglog (h, tol, "-ob",
         h, err, "-b",
         h, (h/h(end)) .^ 4 .* tol(end), "k--",
         h, (h/h(end)) .^ 5 .* tol(end), "k-");
 axis tight
 xlabel ("h");
 ylabel ("err(h)");
 title ("Convergence plot for ode45");
 legend ("imposed tolerance", "ode45 (relative) error",
         "order 4", "order 5", "location", "northwest");
```

```
## Estimate order numerically
p = diff (log (err)) ./ diff (log (h))
```

Conclusión

Hoy en día es inconcebible no ejecutar métodos numéricos en una computadora, software como Octave y Python (con bibliotecas como NumPy y SciPy) tienen una serie de funciones integradas que se pueden usar para una interpolación rápida y precisa, calculo matricial, integración, entre otros. Es decir, en Python, la interpolación se puede implementar fácilmente usando la función interp1d en el paquete SciPy, que admite interpolaciones lineales y polinomiales y es útil en diversas tareas de ingeniería y ciencia; en Octave la función plot grafica líneas, tal que podemos ver el comportamiento de un modelo a través del trazo e curvas.

Por lo tanto, el software hace más eficiente la resolución de tareas complicadas, en particular problemas en el dominio diferencial e integral al manejar conjuntos de datos masivos. Al combinar el uso de métodos numéricos con soluciones de software, los lectores a través de sintaxis sistematizadas en el libro, pueden modelar procesos iterativos en milésimas de segundo. Entre los análisis, destaca el método de Euler y Runge-Kutta para resolver ecuaciones diferenciales, y es a través de Python, en la biblioteca SciPy que se proporcionaron APIs como solve_ivp para soluciones de valor inicial, y en Octave, por otro lado, se ejemplificó la función ode45 con un método subyacente similar para resolver EDOs.

En contexto, la usabilidad de GNU Octave o Python va de acuerdo con las necesidades y cómo y dónde se utilizarán los métodos numéricos. La opción virtuosa y fácil de usar sería GNU Octave. Ahora bien, si se desea más flexibilidad e integración, Python sería la mejor opción. Al final, ambas herramientas ofrecen un conjunto de herramientas para ejecutar métodos numéricos, como fue el caso del desarrollo de Taylor y la diferenciación con Runge-Kutta. La idea central detrás de estos métodos es calcular múltiples pendientes en un intervalo y utilizar estas pendientes para estimar el valor de la función en el siguiente paso. Esto permite una mayor precisión en comparación con métodos más simples, como el método de Euler, que solo utiliza la pendiente en un único punto.

En conclusión, Python tiende a ser más rápido en operaciones complejas y en el manejo de grandes volúmenes de datos, gracias a su capacidad para optimizar el uso de memoria y su diseño orientado a objetos. Por otro lado, Octave puede ser más accesible para quienes están familiarizados con MATLAB, debido a su sintaxis similar. GNU Octave, con su enfoque en la facilidad de uso y su similitud con MATLAB, es una opción excelente para quienes buscan un entorno dedicado al álgebra lineal y a la computación numérica. Finalmente, los métodos numéricos seguirán siendo un componente esencial en la investigación y el desarrollo en la era digital, la combinación de teoría sólida y aplicaciones prácticas en entornos de programación facilitará un avance continuo en la eficacia de estos.

Bibliografía

Abramowitz, M., y Stegun, I.A. (Eds.) (1972). *Manual de Funciones Matemáticas con Fórmulas, Gráficas y Tablas Matemáticas.* Nueva York: Dover

Borrell, G. (2008). *Introducción informal a Matlab y Octave.* Madrid: RedIRIS

Canchoa, A. (2009). Métodos de integración numérica para particiones no uniformes del intervalo de integración. *Anales científicos UNALM,* 70(2), 52-65

Casteleiro, J.M. (2002). *Cálculo integral.* Madrid: Esic Editorial

Çevik, M., Savaşaneril, N.B., y Sezer, M. (2025). Revisión de métodos de colocación de matrices polinómicas en aplicaciones científicas y de ingeniería. *Arch Computat Methods Eng,* 32 , 3355–3373. https://doi.org/10.1007/s11831-025-10235-6

Companioni Guerra, A., Cuesta Llanes, E., Hernández Blanco, Y., Orovio Cobo, V., y Días Ramos, S. (2012). Entorno integrado para el trabajo con GNU/Octave. *Revista Cubana de Ciencias Informáticas,* 6(4), 1-8

Downey, A. (2023). *Modelado y simulación en Python: una introducción para científicos e ingenieros.* San Francisco: No Starch Press

Espinosa Guzmán, A., Espinosa Guzmán, C., Roberto Rodríguez, M.Á. (2016). Comparativo de los Métodos de Mínimos Cuadrados y Eliminación de Gauss-Jordan para la Resolución de Sistema de Ecuaciones en el tema de Regresión Lineal Conciencia Tecnológica, 52, 2016 Instituto Tecnológico de Aguascalientes, México. Disponible en: https://www.redalyc.org/articulo.oa?id=94451204007

Ezquerro, J.A. (2012). *Iniciación a los métodos numéricos.* Logroño: Universidad de La Rioja

Ferreira, I., Acebrón, J., y Monteiro, J. (2025). Estudio de una clase de métodos iterativos de acción por filas: el método Kaczmarz. *Número Algor.,* 99, 2097–2135. https://doi.org/10.1007/s11075-024-01945-2

Gil Iñiguez, J. (2006). Fundamentos sobre convergencia en modelos iterativos para sistemas de ecuaciones lineales. *Fides et Ratio - Revista de Difusión cultural y científica de la Universidad La Salle en Bolivia*, 1(1), 22-32

González, A. (2008). Comparación de métodos analíticos y numéricos para la solución del lanzamiento vertical de una bola en el aire. *Latin-American Journal of Physics Education,* 2(2), 170-179

Guanga Chunata, D.M., Sarmiento Torres, L.E., y Inca Chunata, M. de J. (2019). Métodos numéricos aplicados al conteo de operaciones para el cálculo del Costo computacional en Python. *Ciencia Digital*, *3*(3.3), 331-344. https://doi.org/10.33262/cienciadigital.v3i3.3.825

Linge, S., y Langtangen, H.P. (2016). Cálculo de integrales. En: Programación para cálculos - Python. Textos en ciencia e ingeniería computacional, vol. 15. Springer, Cham. https://doi.org/10.1007/978-3-319-32428-9_3

Lopez, J.H., y Riascos, A.P. (2024). Numfracpy, Técnicas del Cálculo Fraccionario en Python. *Ciencia en Desarrollo*, *15*(2), 43-52

Luna-Fox, S., Caiza Falconí, J., Castelo Naveda, M., y Uvidia Armijo, L. (2024). Aplicaciones del polinomio de Taylor en la resolución de límites indeterminados. *Reincisol*, 3(5), 469-481. https://doi.org/10.59282/reincisol.V3(5)469-481

Mata Rodríguez, C. (2016). Análisis comparativo de los métodos de Euler y Runge-Kutta en la solución numérica de ecuaciones diferenciales de primer orden mediante programación en Mathcad", *Rev. Ing. Mat. Cienc. Inf*, 3(5), http://dx.doi.org/10.21017/rimci.2016.v3.n5.a2

Morales Soto, A., y Cordero Osorio, F. (2014). La graficación - modelación y la Serie de Taylor. Una socioepistemología del cálculo. *Revista latinoamericana de investigación en matemática educativa*, *17*(3), 319-345. https://doi.org/10.12802/relime.13.1733

Nakamura, S. (1992). *Métodos numéricos aplicados con software*. Naucalpan: Prentice Hall Hispanoamericana

Ortega-Laurel, C., y Simental-Franco, V.A. (2021). Metodología de estimación agregada "gruesa" de los ingresos fiscales. *Revista Mexicana de Economía y Finanzas. Nueva Época*, 16(4), 1-16. https://doi.org/10.21919/remef.v16i4.509

Pochulu, M.D. (2018). *La modelización en matemática: marco de referencia y aplicaciones.* Villa María: GIDED

Poole, D. (2006). *Álgebra lineal: Una introducción moderna.* Ciudad de México: Cengage Learning Editores

Raffo Lecca, E., Huatuco, R.M., y Perez Quispe, V. (2007). Algoritmos de integración numérica. Industrial Data, 10(2), 47-53

Riyantoko, P.A., Funabiki, N., Brata, K.C., Mentari, M., Damaliana, A.T., y Prasetya, D.A. (2025). Un método de autoaprendizaje de estadística fundamental con programación en Python para implementaciones de ciencia de datos. *Información* , *16* (7), 607. https://doi.org/10.3390/info16070607

Solís Zúñiga, A.G., Cordero Barbero, A., Torregrosa Sánchez, J.R., y Soto Quirós, J.P. (2021). Diseño y análisis de la convergencia y estabilidad de métodos iterativos para la resolución de ecuaciones no lineales. *Revista Digital: Matemática, Educación E Internet*, *21*(2). https://doi.org/10.18845/rdmei.v21i2.5602

Strang, G. (1980). *Linear algebra and its applications.* London: Academic Press

Thomas, G.B. (2010). *Cálculo. Varias Variables.* Ciudad de México: Pearson

Zota Uño, V. (2015). Efecto de la interpolación respecto al error en una aplicación. *Revista CON-CIENCIA*, *3*(1), 77-86

De esta edición de **“Métodos numéricos aplicados con software: Sintaxis en GNU Octave y Python”**, se terminó de editar en la ciudad de Colonia del Sacramento en la República Oriental del Uruguay el 16 de junio de 2025

EST. 2021 EMC
EDITORIAL MAR CARIBE

Métodos numéricos aplicados con software:

Sintaxis en GNU Octave y Python

2025

ISBN: 978-9915-698-16-8