Uso de programas para la resolución de problemas.

Trabajo Final – Métodos Numéricos.

Daniel Nájera Flores
División de Ingenierías Campus Irapuato - Salamanca, Universidad de Guanajuato.
d.najeraflores@gto.mx

I. INTRODUCCIÓN

La resolución precisa de problemas matemáticos a través de métodos numéricos y el uso de programas en Python representa un desafío fundamental en el ámbito académico y científico. En este estudio, propongo un enfoque basado en técnicas computacionales para abordar diversas problemáticas matemáticas, utilizando programas que he desarrollado mediante la aplicación de métodos numéricos aprendidos durante el semestre.

Los métodos numéricos proporcionan herramientas efectivas para abordar una amplia variedad de problemas matemáticos, permitiendo la cuantificación y caracterización precisa de las soluciones. En este documento, se revisan las técnicas y los métodos más relevantes en el campo de los métodos numéricos, con un enfoque específico en la implementación de programas en Python para resolver ecuaciones, optimizar funciones y abordar diversas problemáticas matemáticas.

II. DESARROLLO DE CONTENIDOS

A lo largo de este trabajo, se explorarán técnicas computacionales, su implementación en Python y se examinarán los resultados obtenidos al aplicar estos métodos para resolver ecuaciones, optimizar funciones y abordar diversas problemáticas matemáticas. Este enfoque ofrece una perspectiva práctica y aplicada, destacando la relevancia de la programación y los métodos numéricos en la resolución efectiva de problemas matemáticos en entornos académicos y científicos.

A. Objetivo del trabajo.

El objetivo principal de este trabajo consiste en aplicar y demostrar la eficacia de los programas desarrollados en el curso de métodos numéricos para resolver problemas matemáticos específicos. El enfoque se centra en abordar los problemas indicados utilizando los métodos numéricos aprendidos durante el semestre y reportar los resultados obtenidos mediante la implementación de programas en Python. Además, se busca proporcionar evidencia clara y detallada de la funcionalidad de los programas, destacando la forma en que se utilizan para la resolución de cada problema matemático planteado.

1) Primer Ejercicio.

Determine gráficamente las raíces reales de las funciones siguientes,

1. $f\,(x) = -26 + 85x - 91x2 + 44x3 - 8x4 + x5$
2. $f(x) = \frac{(0.8-0.3x)}{x}$

2) Segundo Ejercicio.

Sea la función$f(x) = -x^3 - \cos(x)$. Utilizando $P_0 = -1, p_1 = 0$, determine el punto raíz p_3 utilizando el
3. método de la secante,

3) Tercer Ejercicio.

Determine la raíz positiva de la función $\ln(x^4) = 0.7$ empleando 3 iteraciones del método de bisección, con valores iniciales $x_l = 0, x_u = 2$

4) Cuarto Ejercicio.

Aplique: a) el método de Newton-Raphson y b) el método de la secante modificado ($\delta = 0.05$) para determinar una raíz de la función,

$$f(x) = x^5 - 16.05x^4 + 88.75x^3 - 192.0375x^2 + 116.35x + 31.6875$$

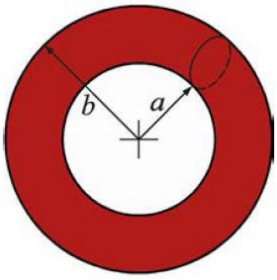
utilizando condición inicial $x_0 = 0.5825$ y tolerancia $\epsilon_s = 10^{-4}$

5) Quinto Ejercicio.

El volumen V de un tubo de agua de forma toroidal está dado por,

$$V = \frac{1}{4}\pi^2(r_1 + r_2)(r_2 - r_1)^2$$

Donde r_1 y r_2 son los radios interno y externo, respectivamente. Determine r_1 si $V = 2500in^2$ y utilizando el método de Newton-Raphson.



6) Sexto Ejercicio.

Resuelva el sistema de ecuaciones no lineal,

$$-2x^3 + 3y^2 + 42 = 0$$
$$5x^2 + 3y^3 - 69 = 0$$

Utilice el método de Newton-Raphson, comenzando con las condiciones x =1, y =1. Realice 5 iteraciones.

7) Séptimo Ejercicio.

Obtenga la solución de x y y para el siguiente sistema de ecuaciones. Considere $x = 3.7, y = 0.25$ como estimaciones iniciales.

$$x^3 - 3x^2y + 2xy^2 + y - 0.596 = 0$$
$$4x^2 - 6.2y^3 + 5.1xy - 6.4319 = 0$$

8) Octavo Ejercicio.

Resuelva los sistemas de ecuaciones siguientes mediante eliminación de Gauss,

$$\begin{aligned} \pi x_1 + \sqrt{2}x_2 - x_3 + x_4 &= 0, \\ ex_1 - x_2 + x_3 + 2x_4 &= 1, \\ x_1 + x_2 - \sqrt{3}x_3 + x_4 &= 2, \\ -x_1 - x_2 + x_3 - \sqrt{5}x_4 &= 3. \end{aligned}$$

$$\begin{bmatrix} 0 & 3 & 8 & -5 & -1 & 6 \\ 3 & 12 & -4 & 8 & 5 & -2 \\ 8 & 0 & 0 & 10 & -3 & 7 \\ 3 & 1 & 0 & 0 & 0 & 4 \\ 0 & 0 & 4 & -6 & 0 & 2 \\ 3 & 0 & 5 & 0 & 0 & -6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 34 \\ 20 \\ 45 \\ 36 \\ 60 \\ 28 \end{bmatrix}$$

9) Noveno Ejercicio.

Una matriz A es no-singular si su determinante es diferente de cero, es decir, $\det|A| \neq 0$. Esta es la condición para que la inversa de la matriz A exista(A^{-1}).
Para las siguientes matrices, compruebe si son no-singulares, y si no lo son, obtenga su inversa.

$$\begin{bmatrix} 4 & 2 & 6 \\ 3 & 0 & 7 \\ -2 & -1 & -3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & -1 & 1 \\ -3 & 2 & 0 & 1 \\ 0 & 5 & 2 & 6 \end{bmatrix} \begin{bmatrix} 2 & 0 & 1 & 2 \\ 1 & 1 & 0 & 2 \\ 2 & -1 & 3 & 1 \\ 3 & -1 & 4 & 3 \end{bmatrix}$$

10) Decimo Ejercicio.

Aplique el método LU y solucione los sistemas de ecuaciones lineales siguientes

$$\begin{aligned} 6x_1 + 2x_2 + x_3 - x_4 &= 0, \\ 2x_1 + 4x_2 + x_3 &= 7, \\ x_1 + x_2 + 4x_3 - x_4 &= -1, \\ -x_1 - x_3 + 3x_4 &= -2. \end{aligned} \qquad \begin{aligned} 2x_1 - x_2 &= 1, \\ x_1 + 2x_2 - x_3 &= 2, \\ 2x_2 + 4x_3 - x_4 &= -1, \\ 2x_4 - x_5 &= -2, \\ x_4 + 2x_5 &= -1. \end{aligned}$$

11) Decimoprimer Ejercicio.

Aplique la regla de Cramer para la solución del sistema de ecuaciones siguientes,

$$\begin{aligned} 2X_1 - 3X_2 + 2X_3 - 4X_4 &= 2.3 \\ -X_1 - 2X_2 + 3X_3 + 2X_4 &= -3.0 \\ 2X_1 + X_2 - 4X_3 - 2X_4 &= 7.7 \\ -3X_1 + 4X_2 - X_3 + X_4 &= -7.0 \end{aligned}$$

12) Duodécimo Ejercicio.

Encuentre las soluciones de los siguientes sistemas de ecuaciones utilizando el método Gauss-Jordan

$$\begin{aligned} 2X_1 - 4X_2 + 6X_3 &= 1.4 \\ 3X_1 + X_2 - 3X_3 &= -9.8 \\ -X_1 + 2X_2 - 3X_3 &= -0.7 \end{aligned} \qquad \begin{aligned} -X_1 + 2X_2 + 3X_3 - 4X_4 &= 1.5 \\ 2X_1 - 3X_2 - X_3 - X_4 &= -12.0 \\ -X_1 - 3X_2 - X_3 + 5X_4 &= -1.5 \\ 3X_1 + 4X_2 + X_3 - 2X_4 &= 5.1 \end{aligned}$$

13) Decimotercer Ejercicio.

Una compañía de alimentos elabora 5 tipos de paquetes de barras snacks de 1.0 lb que tienen las siguientes composiciones y sus costos.

Mix	Peanuts (lb)	Raisins (lb)	Almonds (lb)	Chocolate Chips (lb)	Dried Plums (lb)	Total Cost of Ingredients ($)
A	0.2	0.2	0.2	0.2	0.2	1.44
B	0.35	0.15	0.35	0	0.15	1.16
C	0.1	0.3	0.1	0.1	0.4	1.38
D	0	0.3	0.1	0.4	0.2	1.78
E	0.15	0.3	0.2	0.35	0	1.61

Empleando la información de la tabla,
a. Obtenga un sistema de ecuaciones
b. Resuelva el sistema de ecuaciones que represente el costo por libra de cada uno de los ingredientes

14) Decimocuarto Ejercicio.

Determine los valores característicos y sus vectores característicos asociados de las siguientes matrices mediante el método directo.

$$\begin{bmatrix} 3 & 2 & -1 \\ 1 & -2 & 3 \\ 2 & 0 & 4 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ 2 & 3 & 2 \\ 1 & 1 & 2 \end{bmatrix}$$

15) Decimoquinto Ejercicio.

Aplique el método de QR para la determinación de los valores y vectores característicos de las matrices siguientes,

$$\begin{bmatrix} 1 & 1 & 2 \\ 4 & 1 & 5 \\ 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 10 & 0 & 0 \\ 1 & -3 & -7 \\ 0 & 2 & 6 \end{bmatrix}$$

16) Decimosexto Ejercicio.

Los siguientes datos fueron cuando para la distancia de frenado de un automóvil sobre una carretera mojada fue medida como función de la velocidad v cuando se aplican los frenos,

v (mi/h)	12.5	25	37.5	50	62.5	75
d (ft)	20	59	118	197	299	420

Determine los coeficientes del polinomio cuadrático $d = a_2v^2 + a_1v + a_0$. Hacer el gráfico del polinomio obtenido y de los datos medidos.

17) Decimoséptimo Ejercicio.

La potencia generada por una turbina eólica varía con la velocidad del viento. En un experimento se obtuvieron los datos siguientes.

Velocidad del viento (mph)	14	22	30	38	46	55
Potencia eléctrica (W)	320	490	540	500	480	395

a. Determine el polinomio de Lagrange que pase por los puntos. Utilizando este polinomio calcule la potencia con la velocidad del viento de 35 mph.
b. Determine el polinomio de interpolación de Newton que pasa por los puntos. Utilizando este polinomio determine la potencia a una velocidad del viento de 33 mph.

B. Metodología

La metodología empleada en este estudio se centra en la implementación y demostración práctica de los métodos numéricos estudiados durante el curso. Cada ejercicio presenta desafíos específicos, y para abordarlos de manera efectiva, se diseñaron y ajustaron programas en Python que reflejan la versatilidad y adaptabilidad de los métodos numéricos a diferentes requerimientos.

En este contexto, se proporcionará una detallada exposición de los códigos creados para cada uno de los métodos numéricos requeridos

en los diversos ejercicios. Cabe destacar que, aunque algunos ejercicios comparten métodos similares, los programas han sido adaptados para satisfacer las particularidades y requisitos específicos de cada problema matemático planteado.

1) Primer Ejercicio.

```
def funcion_a(x):
    return -26 + 85*x - 91*x**2 + 44*x**3 - 8*x**4 + x**5

def funcion_b(x):
    return (0.8 - 0.3*x) / x

# Rango de valores para x
x_vals = np.linspace(-10, 10, 400)

# Calcular los valores de y para cada función
y_vals_a = funcion_a(x_vals)
y_vals_b = funcion_b(x_vals)

# Encontrar raíces
raices_a = np.roots([-26, 85, -91, 44, -8, 1])
raices_b = np.roots([-0.3, 0.8])

# Crear gráficos para cada función
plt.figure(figsize=(12, 6))
```

este código genera dos funciones matemáticas, calcula sus valores en un rango dado, encuentra las raíces de estas funciones y luego crea un gráfico para visualizar las funciones junto con sus raíces.

2) Segundo Ejercicio.

Sea la función$f(x) = -x^3 - \cos(x)$. Utilizando $P_0 = -1, p_1 = 0$, determine el punto raíz p_3 utilizando el
4. método de la secante,

```
def f(x):
    return -x**3 - np.cos(x)

def secant_method(p0, p1, tol, max_iter):
    iter_count = 0
    while iter_count < max_iter:
        f_p0 = f(p0)
        f_p1 = f(p1)

        p2 = p1 - f_p1 * (p1 - p0) / (f_p1 - f_p0)

        if abs(p2 - p1) < tol:
            return p2, iter_count + 1

        p0, p1 = p1, p2
        iter_count += 1

    raise Exception("El método de la secante no convergió
después de {} iteraciones".format(max_iter))

# Parámetros iniciales
p0 = -1
p1 = 0
tolerancia = 1e-8
max_iteraciones = 1000

# Llamada al método de la secante
raiz, iteraciones = secant_method(p0, p1, tolerancia,
max_iteraciones)

# Imprimir resultado
print("La raíz aproximada es:", raiz)
print("Número de iteraciones:", iteraciones)
```

Este código implementa el método de la secante para encontrar una raíz de la función $f(x) = -x\text{^}3 - cos(x)$. El método iterativamente mejora una aproximación inicial de la raíz (dada por p0p0 y p1p1) hasta que la diferencia entre iteraciones consecutivas sea menor que una tolerancia especificada. El resultado incluye la raíz aproximada y el número de iteraciones realizadas.

3) Tercer Ejercicio.

```
def f(x):
    return np.log(x**4) - 0.7

def bisection_method(xl, xu, tol, max_iter):
    iter_count = 0
    iterations = []

    while iter_count < max_iter:
        xr = (xl + xu) / 2
        f_xl = f(xl)
        f_xr = f(xr)

        iterations.append(xr)

        if f_xr == 0 or (xu - xl) / 2 < tol:
            return iterations, iter_count + 1

        if f_xl * f_xr < 0:
            xu = xr
        else:
            xl = xr

        iter_count += 1

    raise Exception("El método de bisección no convergió
después de {} iteraciones".format(max_iter))

# Parámetros iniciales
xl = 0.5
xu = 2
tolerancia = 1e-8
max_iteraciones = 1000  # Aumentar el número máximo de
iteraciones

# Llamada al método de bisección
iterations, iteraciones = bisection_method(xl, xu,
tolerancia, max_iteraciones)
```

Este código implementa el método de la bisección para encontrar una raíz de la función $f(x) = log(x^4) - 0.7$. El método divide iterativamente un intervalo inicial [xl,xu] por la mitad y selecciona el subintervalo que contiene la raíz. El proceso continúa hasta que la diferencia entre los extremos del intervalo sea menor que una tolerancia especificada. El resultado incluye una lista de las aproximaciones intermedias y el número de iteraciones realizadas.

4) Cuarto Ejercicio.

```
# Definir la variable simbólica
x = sp.symbols('x')

# Definir la función f(x)
f = x**5 - 16.05*x**4 + 88.75*x**3 - 192.0375*x**2 +
116.35*x + 31.6875

# Derivar la función f(x)
df = sp.diff(f, x)

# Convertir la función f(x) y su derivada df(x) a funciones
numéricas
f_numeric = sp.lambdify(x, f, 'numpy')
df_numeric = sp.lambdify(x, df, 'numpy')

def newton_raphson_method(x0, tol, max_iter):
    x_values = [x0]
    iter_count = 0

    while iter_count < max_iter:
        x_new = x_values[-1] - f_numeric(x_values[-1]) /
df_numeric(x_values[-1])
        x_values.append(x_new)

        if abs(x_new - x_values[-2]) < tol:
            return x_values, iter_count + 1

        iter_count += 1

    return x_values, iter_count + 1

def modified_secant_method(x0, delta, tol, max_iter):
```

```
    x_values = [x0]
    iter_count = 0

    while iter_count < max_iter:
        f_x = f_numeric(x_values[-1])
        df_x = df_numeric(x_values[-1])

        if abs(df_x) < 1e-10:
            return x_values, iter_count + 1

        x_new = x_values[-1] - delta * f_x / df_x
        x_values.append(x_new)

        if abs(x_new - x_values[-2]) < tol:
            return x_values, iter_count + 1

        iter_count += 1

    return x_values, iter_count + 1

# Parámetros iniciales
x0 = 0.5825
tolerancia = 1e-4
max_iteraciones = 100
delta_secante = 0.05
```

En el código se utiliza la biblioteca SymPy para definir una función simbólica, derivarla y luego convierte estas expresiones simbólicas en funciones numéricas. Luego implementa dos métodos numéricos: el método de Newton-Raphson y el método de la secante modificada. Estos métodos buscan encontrar raíces de la función dada.

5) Quinto Ejercicio.

```
def volumen_toroide(r1, r2):
    return (1/4) * (np.pi**2) * (r1 + r2) * (r2 - r1)**2

def derivada_volumen_toroide(r1, r2):
    return (1/2) * (np.pi**2) * (r1 + r2) * (r1 - r2)

def newton_raphson(volumen_deseado, r2, tolerancia=1e-6,
max_iter=100):
    r1 = r2 / 2  # Asumir un valor inicial para r1

    for i in range(max_iter):
        volumen_actual = volumen_toroide(r1, r2)
        derivada_actual = derivada_volumen_toroide(r1, r2)

        if abs(volumen_actual - volumen_deseado) <
tolerancia:
            print(f"Se alcanzó la convergencia después de
{i+1} iteraciones.")
            return r1

        r1 = r1 - (volumen_actual - volumen_deseado) /
derivada_actual

    print("El método de Newton-Raphson no convergió después
de", max_iter, "iteraciones.")
    return None

# Parámetros dados en el problema
volumen_deseado = 2500  # in^3
r2 = 18  # in

# Calcular r1 usando el método de Newton-Raphson
resultado = newton_raphson(volumen_deseado, r2)

# Imprimir resultado
if resultado is not None:
    print("El valor de r1 es:", resultado)
else:
    print("No se pudo encontrar una solución con el método
de Newton-Raphson.")
```

Aquí se implementa el método de Newton-Raphson para encontrar el radio r1r1 de un toroide (anillo) dado un volumen deseado y el radio r2r2 del toroide. La función volumen toroide calcula el volumen del toroide, **derivada_volumen_toroide** calcula la derivada del volumen con respecto a r1r1, y newton_raphson utiliza el método de Newton-Raphson para encontrar un valor de r1r1 que produce el volumen deseado. Los parámetros iniciales dados son un volumen deseado de 2500 in^3 y un radio r2r2 de 18 pulgadas.

6) Sexto Ejercicio.

```
def sistema_ecuaciones(x, y):
    ecuacion1 = -2 * x**3 + 3 * y**2 + 42
    ecuacion2 = 5 * x**2 + 3 * y**3 - 69
    return np.array([ecuacion1, ecuacion2])

def jacobiano(x, y):
    j11 = -6 * x**2
    j12 = 6 * y
    j21 = 10 * x
    j22 = 9 * y**2
    return np.array([[j11, j12], [j21, j22]])

def newton_raphson(x0, y0, iteraciones):
    solucion = np.array([x0, y0], dtype=float)
    historial_solucion = [solucion.copy()]

    for i in range(iteraciones):
        f = sistema_ecuaciones(solucion[0], solucion[1])
        J = jacobiano(solucion[0], solucion[1])

        delta = np.linalg.solve(J, -f)
        solucion += delta

        historial_solucion.append(solucion.copy())
        print(f"Iteración {i + 1}: x = {solucion[0]}, y =
{solucion[1]}")

    return np.array(historial_solucion)

# Condiciones iniciales
x_inicial = 1
y_inicial = 1
iteraciones = 5

historial_solucion = newton_raphson(x_inicial, y_inicial,
iteraciones)
```

El código implementa el método de Newton-Raphson para resolver un sistema de dos ecuaciones no lineales. Se define el sistema de ecuaciones, se calcula el jacobiano, y se utiliza el método de Newton-Raphson para iterativamente actualizar una solución inicial hasta alcanzar convergencia. El historial de soluciones se registra en cada iteración.

7) Séptimo Ejercicio.

```
def sistema_ecuaciones(vars):
    x, y = vars
    eq1 = x**3 - 3*x**2*y + 2*x*y**2 + y - 0.596
    eq2 = 4*x**2 - 6.2*y**3 + 5.1*x*y - 6.4319
    return [eq1, eq2]

# Estimaciones iniciales
estimaciones_iniciales = [3.7, 0.25]

# Resolviendo el sistema de ecuaciones no lineales
soluciones = fsolve(sistema_ecuaciones,
estimaciones_iniciales)

# Mostrando los resultados
x_solucion, y_solucion = soluciones
print(f"Solución para x: {x_solucion}")
print(f"Solución para y: {y_solucion}")
```

utiliza la función fsolve de la biblioteca SciPy para encontrar las soluciones de un sistema de dos ecuaciones no lineales. El sistema de ecuaciones está definido en la función sistema_ecuaciones, y se proporcionan estimaciones iniciales para las variables xx e yy. El resultado es la solución del sistema, que se imprime al final del código.

8) Octavo Ejercicio.

```
def eliminacion_gauss(A, b):
    n = len(b)

    # Verificar si la matriz es singular
    if np.linalg.matrix_rank(A) < n:
        raise ValueError("La matriz es singular y el
sistema puede no tener solución única.")

    # Combinar matriz A y vector b en una matriz aumentada
    sistema_ecuaciones =
np.column_stack((A.astype(np.float64),
b.astype(np.float64)))

    # Eliminación hacia adelante
    for i in range(n):
        # Hacer que el pivote sea igual a 1
        sistema_ecuaciones[i, :] /= sistema_ecuaciones[i,
i]

        # Eliminar otros elementos en la columna actual
        for j in range(i + 1, n):
            sistema_ecuaciones[j, :] -=
sistema_ecuaciones[j, i] * sistema_ecuaciones[i, :]

    # Sustitución hacia atrás
    x = np.zeros(n)
    for i in range(n - 1, -1, -1):
        x[i] = sistema_ecuaciones[i, -1] -
np.dot(sistema_ecuaciones[i, i+1:n], x[i+1:])

    return x

# Sistema de ecuaciones 1
A1 = np.array([
    [np.pi, np.sqrt(2), -1, 1],
    [np.e, -1, 1, 2],
    [1, 1, -np.sqrt(3), 1],
    [-1, -1, 1, -np.sqrt(5)]
])
b1 = np.array([0, 1, 2, 3])

# Sistema de ecuaciones 2
A2 = np.array([
    [0, 3, 8, -5, -1, 6],
    [3, 12, -4, 8, 5, -2],
    [8, 0, 0, 10, -3, 7],
    [3, 1, 0, 0, 0, 4],
    [0, 0, 4, -6, 0, 2],
    [3, 0, 5, 0, 0, -6]
])
b2 = np.array([34, 20, 45, 36, 60, 28])
```

La función **eliminacion_gauss** realiza la eliminación hacia adelante y la sustitución hacia atrás para encontrar la solución de un sistema de ecuaciones lineales **Ax=b**. Se proporcionan dos ejemplos de sistemas lineales, y las soluciones se imprimen al final del código.

9) Noveno Ejercicio.

```
def es_no_singular(matriz):
    determinante = np.linalg.det(matriz)
    return determinante != 0

def obtener_inversa(matriz):
    if es_no_singular(matriz):
        inversa = np.linalg.inv(matriz)
        return inversa
    else:
        return None

# Matriz 1
matriz1 = np.array([
    [4, 2, 6],
    [3, 0, 7],
    [-2, -1, -3]
])

# Matriz 2
matriz2 = np.array([
    [1, 2, 3, 4],
    [2, 1, -1, 1],
    [-3, 2, 0, 1],
    [0, 5, 2, 6]
])

# Matriz 3
matriz3 = np.array([
    [2, 0, 1, 2],
    [1, 1, 0, 2],
    [2, -1, 3, 1],
    [3, -1, 4, 3]
])

# Verificar si son no-singulares y obtener inversas si es
posible
print("Matriz 1:")
if es_no_singular(matriz1):
    inversa1 = obtener_inversa(matriz1)
    print(f"La matriz es no-singular.
Inversa:\n{inversa1}")
else:
    print("La matriz es singular y no tiene inversa.")

print("\nMatriz 2:")
if es_no_singular(matriz2):
    inversa2 = obtener_inversa(matriz2)
    print(f"La matriz es no-singular.
Inversa:\n{inversa2}")
else:
    print("La matriz es singular y no tiene inversa.")

print("\nMatriz 3:")
if es_no_singular(matriz3):
    inversa3 = obtener_inversa(matriz3)
    print(f"La matriz es no-singular.
Inversa:\n{inversa3}")
else:
    print("La matriz es singular y no tiene inversa.")
```

El código verifica si tres matrices dadas son no-singulares y, en caso afirmativo, calcula e imprime sus inversas. Utiliza funciones para determinar la no-singularidad y calcular la inversa, y luego muestra los resultados para cada matriz.

10) Decimo Ejercicio.

```
def resolver_sistema_lu(A, b):
    # Obtener la descomposición LU con pivoteo parcial
    p, lu_matrix = lu(A, permute_l=True)

    # Descomponer la matriz LU
    l = np.tril(lu_matrix, k=-1) + np.eye(len(A))
    u = np.triu(lu_matrix)

    # Resolver Ly = Pb usando sustitución hacia adelante
    y = solve(l, np.dot(p, b), assume_a="gen")

    # Resolver Ux = y usando sustitución hacia atrás
    x = solve(u, y, assume_a="gen")

    return l, u, x

# Sistema 1
A1 = np.array([
    [6, 2, 1, -1],
    [2, 4, 1, 0],
    [1, 1, 4, -1],
    [-1, 0, -1, 3]
])
b1 = np.array([0, 7, -1, -2])

l1, u1, solucion_sistema1 = resolver_sistema_lu(A1, b1)
print("Matriz L del sistema 1:")
print(l1)
print("\nMatriz U del sistema 1:")
print(u1)
print("\nSolución del sistema 1:")
print(solucion_sistema1)

# Sistema 2
A2 = np.array([
    [2, -1, 0, 0, 0],
    [1, 2, -1, 0, 0],
    [0, 2, 4, -1, 0],
    [0, 0, 0, 2, -1],
    [0, 0, 0, 1, 2]
```

```
])
b2 = np.array([1, 2, -1, -2, -1])

l2, u2, solucion_sistema2 = resolver_sistema_lu(A2, b2)
```

El código mostrado resuelve sistemas de ecuaciones lineales mediante la descomposición LU con pivoteo parcial. Utiliza la función **resolver_sistema_lu**, que obtiene las matrices LL y UU de la descomposición LU y resuelve el sistema de ecuaciones lineales. Luego, se aplica a dos sistemas específicos y se imprimen las matrices LL y UU junto con las soluciones.

11) Decimoprimer Ejercicio.

```
def regla_de_cramer(matriz_coeficientes,
vector_terminos_independientes):
    det_A = np.linalg.det(matriz_coeficientes)
    n = len(matriz_coeficientes)

    soluciones = []

    for i in range(n):
        matriz_sustituida = matriz_coeficientes.copy()
        matriz_sustituida[:, i] =
vector_terminos_independientes
        det_Ai = np.linalg.det(matriz_sustituida)
        solucion_i = det_Ai / det_A
        soluciones.append(solucion_i)

    return soluciones

# Sistema de ecuaciones
A = np.array([
    [2, -3, 2, -4],
    [-1, -2, 3, 2],
    [2, 1, -4, -2],
    [-3, 4, -1, 1]
])

b = np.array([2.3, -3.0, 7.7, -7.0])

# Resolver utilizando la regla de Cramer
soluciones = regla_de_cramer(A, b)
```

Se implementa la regla de **Cramer** para resolver un sistema de ecuaciones lineales. Calcula el determinante de la matriz de coeficientes y, para cada variable, sustituye la columna correspondiente con el vector de términos independientes y calcula el determinante. Las soluciones se obtienen como cocientes de determinantes y se almacenan en una lista. En este ejemplo, se aplica la regla de Cramer a un sistema específico y se almacenan las soluciones en la variable soluciones.

12) Duodécimo Ejercicio.

```
def gauss_jordan(A, b):
    matriz_aumentada = np.column_stack((A, b))
    m, n = matriz_aumentada.shape

    for i in range(m):
        # Normalizar la fila actual
        matriz_aumentada[i, :] /= matriz_aumentada[i, i]

        # Eliminación hacia adelante
        for j in range(m):
            if i != j:
                matriz_aumentada[j, :] -=
matriz_aumentada[j, i] * matriz_aumentada[i, :]

    # Extraer la solución del sistema
    soluciones = matriz_aumentada[:, -1]

    return soluciones

# Sistema 1
A1 = np.array([
    [2, -4, 6],
    [3, 1, -3],
    [-1, -2, -3]
])
b1 = np.array([1.4, -9.8, -0.7])

soluciones_sistema1 = gauss_jordan(A1, b1)
print("Soluciones del sistema 1:")
print(soluciones_sistema1)

# Sistema 2
A2 = np.array([
    [-1, 2, 3, -4],
    [2, -3, -1, -1],
    [-1, -3, -1, 5],
    [3, 4, 1, -2]
])
b2 = np.array([1.5, -12.0, -1.5, 5.1])

soluciones_sistema2 = gauss_jordan(A2, b2)
```

El código implementa el método de **Gauss-Jordan** para resolver sistemas de ecuaciones lineales. Combina la matriz de coeficientes y el vector de términos independientes en una matriz aumentada, realiza eliminación hacia adelante y sustitución hacia atrás para obtener una forma escalonada reducida por filas, y extrae las soluciones del sistema. Se aplica este método a **dos sistemas específicos** y **se imprimen las soluciones**.

13) Decimotercer Ejercicio.

```
# Composiciones y costos de los paquetes de barras snacks
compositions = np.array([
    [0.2, 0.2, 0.2, 0.2, 0.2],
    [0.35, 0.15, 0.35, 0, 0.15],
    [0.1, 0.3, 0.1, 0.1, 0.4],
    [0, 0.3, 0.1, 0.4, 0.2],
    [0.15, 0.3, 0.2, 0.35, 0]
])

costs = np.array([1.44, 1.16, 1.38, 1.78, 1.61])

ingredient_names = ['Peanuts', 'Raisins', 'Almonds',
'Chocolate Chips', 'Dried Plums']

# Imprime el sistema de ecuaciones
print("Sistema de ecuaciones (Ax = B):")
for i in range(compositions.shape[0]):
    equation = " + ".join([f"{compositions[i,
j]:.2f}({ingredient_names[j]})" for j in
range(compositions.shape[1])])
    print(f"{equation} = {costs[i]}")

# Resuelve el sistema de ecuaciones lineales
solution = np.linalg.solve(compositions, costs)

# Imprime el resultado
print("\nCosto por libra de cada ingrediente:")
for i in range(len(solution)):
    print(f"{ingredient_names[i]} ({'x_'+str(i+1)}):
${solution[i]:.2f}")
```

Para el ejercicio dado se **modela y resuelve un sistema de ecuaciones** lineales que representa la composición y costos de varios ingredientes en paquetes de barras snacks. Utiliza la función **np.linalg.solve** para encontrar los costos por libra de cada ingrediente y luego imprime los resultados.

14) Decimocuarto Ejercicio.

```
# Matriz 1
matrix1 = np.array([[3, 2, -1],
                    [1, -2, 3],
                    [2, 0, 4]])

# Matriz 2
matrix2 = np.array([[2, 1, 1],
                    [2, 3, 2],
                    [1, 1, 2]])

# Función para obtener los valores característicos y
vectores característicos
def obtener_caracteristicas(matriz):
    eigenvalues, eigenvectors = np.linalg.eig(matriz)
    return eigenvalues, eigenvectors
```

```
# Obtener los valores característicos y vectores
característicos para la Matriz 1
eigenvalues1, eigenvectors1 =
obtener_caracteristicas(matrix1)

# Obtener los valores característicos y vectores
característicos para la Matriz 2
eigenvalues2, eigenvectors2 =
obtener_caracteristicas(matrix2)
```

Para este caso se calculan los **valores característicos y vectores característicos** de dos matrices utilizando la función **np.linalg.eig** en Python. Las matrices y sus características se almacenan en variables correspondientes.

15) Decimoquinto Ejercicio.

```
# Definir las matrices
matrix1 = np.array([[1, 1, 2],
                    [4, 1, 5],
                    [0, 0, 1]])

matrix2 = np.array([[10, 0, 0],
                    [1, -3, -7],
                    [0, 2, 6]])

# Función para aplicar el método QR
def qr_method(matriz, max_iter=1000, tol=1e-10):
    n = matriz.shape[0]
    A = np.copy(matriz)

    for _ in range(max_iter):
        Q, R = np.linalg.qr(A)
        A = R @ Q

        # Verificar convergencia
        if np.linalg.norm(np.triu(A, k=1)) < tol:
            break

    # Extraer valores y vectores característicos
    eigenvalues = np.diagonal(A)
    eigenvectors = np.eye(n)

    for i in range(n):
        for j in range(max_iter):
            eigenvectors[:, i] = Q @ eigenvectors[:, i]

    return eigenvalues, eigenvectors

# Aplicar el método QR a la Matriz 1
eigenvalues1, eigenvectors1 = qr_method(matrix1)

# Aplicar el método QR a la Matriz 2
eigenvalues2, eigenvectors2 = qr_method(matrix2)
```

El código utiliza el método **QR** para calcular los valores y vectores característicos de dos matrices en Python. La función **qr_method** aplica iterativamente el método QR hasta que la matriz converja o se alcance el número máximo de iteraciones, y luego extrae los valores y **calcula los vectores característicos**.

16) Decimosexto Ejercicio.

```
# Datos medidos
velocidades = np.array([12.5, 25, 37.5, 50, 62.5, 75])
distancias = np.array([20, 59, 118, 197, 299, 420])

# Ajuste de mínimos cuadrados para un polinomio cuadrático
coefficients = np.polyfit(velocidades, distancias, 2)

# Coeficientes del polinomio cuadrático
a2, a1, a0 = coefficients

# Polinomio cuadrático
polynomial = np.poly1d(coefficients)

# Valores para graficar el polinomio
velocidades_fit = np.linspace(min(velocidades),
max(velocidades), 100)
distancias_fit = polynomial(velocidades_fit)
```

Se realiza un ajuste de mínimos cuadrados a datos de velocidades y distancias, utilizando un **polinomio cuadrático**. Luego, extrae los coeficientes del polinomio ajustado y genera valores para **graficar la curva resultante**.

17) Decimoséptimo Ejercicio.

```
# Datos del experimento
velocidades = np.array([14, 22, 30, 38, 46, 55])
potencia_electrica = np.array([320, 490, 540, 500, 480,
395])

# Método de interpolación de Lagrange
def lagrange_interpolation(x, x_values, y_values):
    result = 0
    n = len(x_values)
    for i in range(n):
        term = y_values[i]
        for j in range(n):
            if j != i:
                term = term * (x - x_values[j]) /
(x_values[i] - x_values[j])
        result += term
    return result

# Método de interpolación de Newton
def newton_interpolation(x, x_values, y_values):
    n = len(x_values)
    coefficients = y_values.copy()

    for j in range(1, n):
        for i in range(n-1, j-1, -1):
            coefficients[i] = (coefficients[i] -
coefficients[i-1]) / (x_values[i] - x_values[i-j])

    result = coefficients[-1]
    for i in range(n-2, -1, -1):
        result = result * (x - x_values[i]) +
coefficients[i]

    return result

# a. Polinomio de Lagrange y potencia a 35 mph
velocidad_evaluar_a = 35
potencia_lagrange_a =
lagrange_interpolation(velocidad_evaluar_a, velocidades,
potencia_electrica)
print(f"a. Potencia a {velocidad_evaluar_a} mph (Lagrange):
{potencia_lagrange_a} W")

# b. Polinomio de Newton y potencia a 33 mph
velocidad_evaluar_b = 33
potencia_newton_b =
newton_interpolation(velocidad_evaluar_b, velocidades,
potencia_electrica)
print(f"b. Potencia a {velocidad_evaluar_b} mph (Newton):
{potencia_newton_b} W")
```

En el cálculo realiza **interpolación** de datos experimentales de velocidades y potencias eléctricas. Utiliza los métodos de interpolación de **Lagrange y Newton** para estimar la potencia a velocidades específicas, luego imprime los resultados.

IV. RESULTADOS

En esta sección, se presentarán los resultados obtenidos de manera **numérica** y, cuando aplicable, de forma **gráfica**, proporcionando una visión integral de la eficacia de los métodos implementados.

Cada ejercicio se analizará detalladamente, revelando los valores numéricos obtenidos mediante la ejecución de los programas diseñados. Para aquellos problemas donde la visualización resulta esencial, se incluirán representaciones gráficas que complementen y enriquezcan la interpretación de los resultados. Es importante señalar que, aunque **la utilización de muestras gráficas puede no ser necesaria en todos los ejercicios**, se incorporarán cuando contribuyan significativamente a la comprensión y validación de las soluciones propuestas.

1) Primer Ejercicio.

```
Raíces de la función a:
[1.7952499 +0.j; 0.63872964+0.39608584j
0.63872964-0.39608584j; 0.09826079+0.16814557j;
0.09826079-0.16814557j]

    Raíces de la función b: [2.66666667]
```

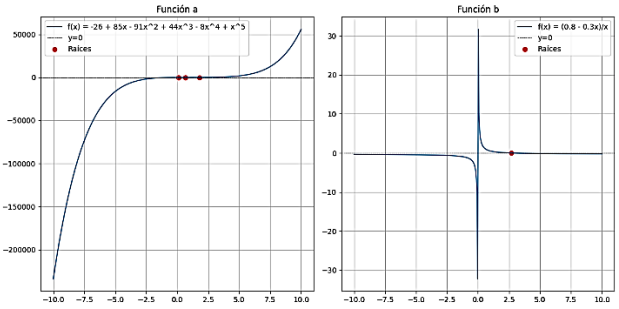


Fig. 1. Graficas de los resultados de las raices.

2) Segundo Ejercicio.

```
La raíz aproximada es: -0.8654740331016144
Número de iteraciones: 9
```

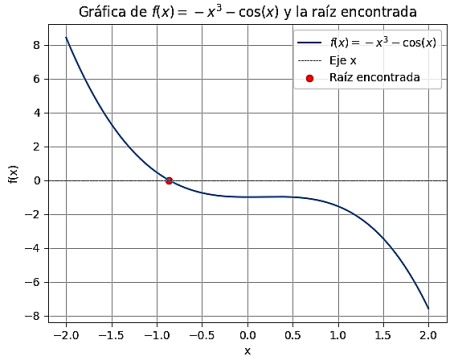


Fig. 2. Resultado de la raizaproximada.

3) Tercer Ejercicio.

```
La raíz positiva aproximada es:
1.1912462171167135
Número de iteraciones: 28
```

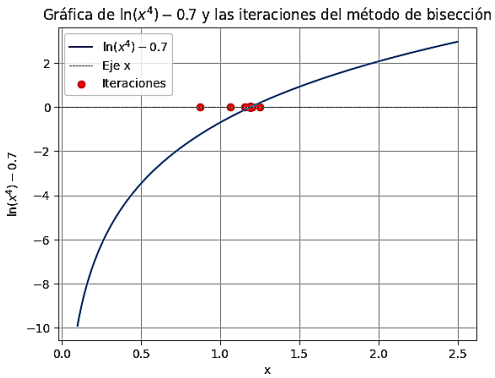


Fig. 3. Resultado mediante el método de bisección.

4) Cuarto Ejercicio.

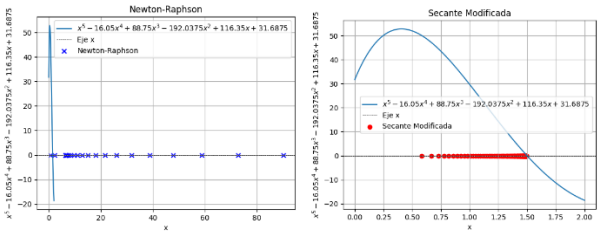


Fig. 4. Resultados del cálculo.

Como se puede apreciar en la gráfica, los resultados obtienen múltiples valores a lo largo de las iteraciones, el método empleado para Newton-Raphson es el que nos arroja mejores resultados, pero, el método de la secante modificada no convergió después de 100 iteraciones.

5) Quinto Ejercicio.

```
Se alcanzó la convergencia después de 11
iteraciones.
El valor de r1 es: 12.208581323757976
```

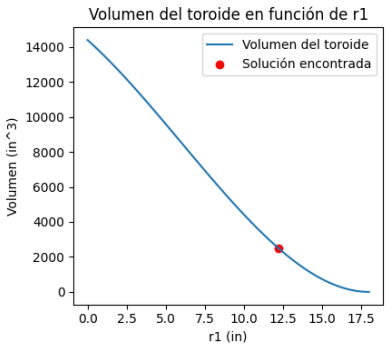


Fig. 5. Resultado del cálculo del toroide.

6) Sexto Ejercicio.

```
Iteración 1:
x = 7.605263157894735,
y = 0.438596491228071
Iteración 2:
x = 5.071351939787781,
y = -15.585550350178893
Iteración 3:
x = 5.246202608486081,
y = -10.421681891076977
Iteración 4:
x = 4.4192882179587105,
y = -6.97360022937365
Iteración 5:
x = 3.7177018196257845,
y = -4.643687127104711
```

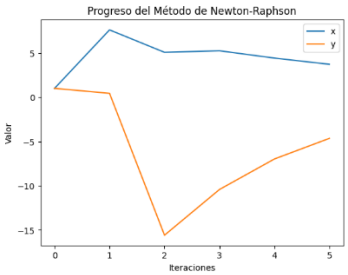


Fig. 6. Resultado por Newton-Raphson.

7) Séptimo Ejercicio.

```
Solución para x: 1.0750890296732676
Solución para y: 0.4049608192296603
```

8) Octavo Ejercicio.

```
Solución del sistema de ecuaciones 1:
[ 1.34944856 -4.67798775 -4.03289378 -
1.65663773]

Solución del sistema de ecuaciones 2:
[ -67.15763547 -284.93103448  202.61576355
168.61083744  672.68472906 130.60098522]
```

9) Noveno Ejercicio.

```
Matriz 1:
La matriz es singular y no tiene inversa.

Matriz 2:
La matriz es singular y no tiene inversa.

Matriz 3:
La matriz es no-singular. Inversa:
[[ 1.00000000e+00  7.40148683e-17  1.00000000e+00 -
1.00000000e+00]
 [-1.00000000e+00  1.66666667e+00  1.66666667e+00 -
1.00000000e+00]
 [-1.00000000e+00  6.66666667e-01  6.66666667e-01 -
0.00000000e+00]
 [-0.00000000e+00 -3.33333333e-01 -1.33333333e+00
1.00000000e+00]]
```

10) Decimo Ejercicio.

```
Matriz L del sistema 1:
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]

Matriz U del sistema 1:
[[ 6.          2.          1.         -1.        ]
 [ 0.          3.33333333  0.66666667  0.33333333]
 [ 0.          0.          3.7        -0.9       ]
 [ 0.          0.          0.          2.58108108]]

Solución del sistema 1:
[-0.78273666  2.13923872  0.00851847 -0.40942408 ]

Matriz L del sistema 2:
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]

Matriz U del sistema 2:
[[ 2.  -1.   0.   0.   0. ]
 [ 0.   2.5 -1.   0.   0. ]
 [ 0.   0.   4.8 -1.   0. ]
 [ 0.   0.   0.   2.  -1. ]
 [ 0.   0.   0.   0.   2.5]]

Solución del sistema 2:
[ 0.96666667  0.93333333 -0.16666667 -1.4    -0.8 ]
```

11) Decimoprimer Ejercicio.

```
Soluciones del sistema:
    x1 = 1.4999999999999976
    x2 = -1.0666666666666662
    x3 = -1.4333333333333325
    x4 = 0.33333333333333337
```

12) Duodécimo Ejercicio.

```
Soluciones del sistema 1:
[-2.275       0.          0.99166667]

Soluciones del sistema 2:
[-1.3  2.9 -0.4  1.1]
```

13) Decimotercer Ejercicio.

```
Sistema de ecuaciones (Ax = B):
0.20(Peanuts) + 0.20(Raisins) + 0.20(Almonds) +
0.20(Chocolate Chips) + 0.20(Dried Plums) =
1.44

0.35(Peanuts) + 0.15(Raisins) + 0.35(Almonds) +
0.00(Chocolate Chips) + 0.15(Dried Plums) =
1.16

0.10(Peanuts) + 0.30(Raisins) + 0.10(Almonds) +
0.10(Chocolate Chips) + 0.40(Dried Plums) =
1.38

0.00(Peanuts) + 0.30(Raisins) + 0.10(Almonds) +
0.40(Chocolate Chips) + 0.20(Dried Plums) =
1.78

0.15(Peanuts) + 0.30(Raisins) + 0.20(Almonds) +
0.35(Chocolate Chips) + 0.00(Dried Plums) =
1.61

Costo por libra de cada ingrediente:
    Peanuts (x_1): $0.35
    Raisins (x_2): $1.14
    Almonds (x_3): $1.85
    Chocolate Chips (x_4): $2.41
    Dried Plums (x_5): $1.44
```

14) Decimocuarto Ejercicio.

```
Matriz 1:
Valores Característicos: [ 3.  4. -2.]
Vectores Característicos:
[[-4.08248290e-01 -1.81298661e-16  3.48742916e-01]
 [ 4.08248290e-01  4.47213595e-01 -9.29981110e-01]
 [ 8.16496581e-01  8.94427191e-01 -1.16247639e-01]]

Matriz 2:
Valores Característicos: [1. 5. 1.]
Vectores Característicos:
[[-0.80178373  0.40824829 -0.29474362]
 [ 0.53452248  0.81649658 -0.51205571]
 [ 0.26726124  0.40824829  0.80679933]]
```

15) Decimoquinto Ejercicio.

```
Matriz 1:
Valores Característicos: [ 3. -1.  1.]
```

```
Vectores Característicos:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

Matriz 2:
Valores Característicos: [10.  4. -1.]
Vectores Característicos:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

16) Decimosexto Ejercicio.

```
Coeficientes del polinomio cuadrático:
a2: 0.0665
a1: 0.5777
a0: 2.6000
```

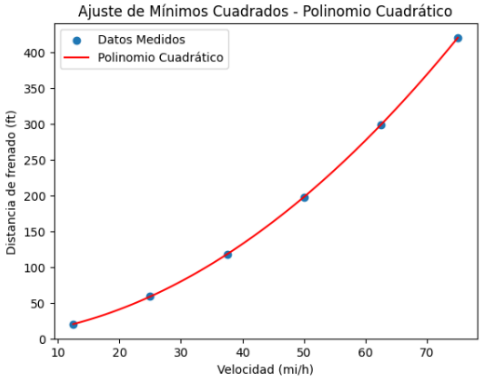


Fig. 7. Muestra gráfica del polinomio couadrático.

17) Decimoséptimo Ejercicio.

```
a. Potencia a 35 mph (Lagrange): 517.4717698245457 W
b. Potencia a 33 mph (Newton): 719 W
```

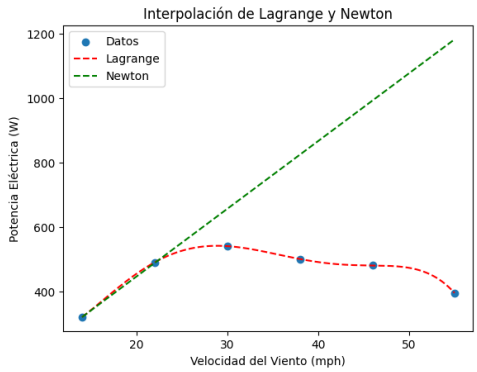


Fig. 8. Muestra de resultados en ambos métodos.

V. CONCLUSIONES.

En conclusión, a lo largo de este trabajo se ha abordado con éxito la resolución de problemas matemáticos mediante la aplicación de métodos numéricos y la implementación de programas en Python. La adaptabilidad de estos métodos se evidenció a través de la creación de códigos específicos para cada ejercicio, cumpliendo con los requisitos y particularidades de cada problema.

Los resultados numéricos y, en algunos casos, las representaciones gráficas han proporcionado una visión completa de la eficacia de los programas desarrollados. La metodología empleada no solo destacó la aplicación de los métodos numéricos aprendidos durante el semestre, sino también resaltó la importancia de la programación en Python como herramienta integral en la resolución de problemas matemáticos complejos.

Los desafíos encontrados en el camino y las soluciones propuestas ofrecen una valiosa contribución al campo de los métodos numéricos, brindando perspectivas prácticas y aplicadas.