Renato Ramírez Sosa

1) Calcular la primera y segunda derivada de las siguientes funciones en x=3.5

a)

```python
import math

# Función y su segunda derivada
def f(x):
    return math.exp(2 * x)

def primera_derivada(x):
    return 2 * math.exp(2 * x)

def segunda_derivada(x):
    return 4 * math.exp(2 * x)

def cuarta_derivada(x):
    return 16 * math.exp(2 * x)

# Valores
x0 = 3.5
h = 0.0001

dif_formula = (f(x0 + h) - f(x0)) / h
dif_formula_2 = (f(x0 + 2*h) - 2*f(x0 + h) + f(x0)) / (h**2)

#Error
cota_error = (h / 2) * abs(segunda_derivada(x0))
cota_error_2 = (h**2 / 12) * abs(cuarta_derivada(x0))

print(f"Derivada 2aproximada en x0 = {x0}: {dif_formula:.6f}")
print(f"Derivada 4 aproximada en x0 = {x0}: {dif_formula_2:.6f}")
print(f"Error 2 derivada: {cota_error:.6f}")
print(f"Error 4 derivada: {cota_error_2:.6f}")
```

✓ 0.0s

```
Derivada aproximada en x0 = 3.5: 2193.485658
Derivada 2 aproximada en x0 = 3.5: 4387.409922
Error: 0.219327
Error: 0.000015
```

b)

```python
import math

# Función y derivadas exactas
def f(x):
    return x * math.cos(x) - x**2 * math.sin(x)

def primera_derivada(x):
    return math.cos(x) - 3*x * math.sin(x) - x**2 * math.cos(x)

def segunda_derivada(x):
    return -4 * math.sin(x) - 5 * x * math.cos(x) + x**2 * math.sin(x)

def cuarta_derivada(x):
    return abs(x**2 * math.sin(x) + 10 * math.sin(x) + 10 * x * math.cos(x) + x**4 * math.sin(x)) + 1

# Valores
x0 = 3.5
h = 0.001

dif_formula = (f(x0 + h) - f(x0)) / h
dif_formula_2 = (f(x0 + 2*h) - 2*f(x0 + h) + f(x0)) / (h**2)

# Error
cota_error = (h / 2) * abs(segunda_derivada(x0))
cota_error_2 = (h**2 / 12) * abs(cuarta_derivada(x0))


print(f"Derivada 2aproximada en x0 = {x0}: {dif_formula:.6f}")
print(f"Derivada 4 aproximada en x0 = {x0}: {dif_formula_2:.6f}")
print(f"Error 2 derivada: {cota_error:.6f}")
print(f"Error 4 derivada: {cota_error_2:.6f}")
```

✓ 0.0s

```
Derivada 2aproximada en x0 = 3.5: 14.225107
Derivada 4 aproximada en x0 = 3.5: 13.482375
Error 2 derivada: 0.006747
Error 4 derivada: 0.000008
```

2)

```python
import numpy as np

# Datos dados
t_vals = np.array([1.00, 1.01, 1.02, 1.03, 1.04, 1.05])  # s
i_vals = np.array([3.100, 3.110, 3.120, 3.140, 3.180, 3.240])  # A

# Constantes del circuito
L = 0.98    # H
R = 0.142   # Ohm

di_dt = []

for k in range(5):  # t = 1.00 a 1.04
    if k == 0:  # hacia adelante
        dt = t_vals[k+1] - t_vals[k]
        deriv = (i_vals[k+1] - i_vals[k]) / dt
    elif k == 4:  # hacia atrás
        dt = t_vals[k] - t_vals[k-1]
        deriv = (i_vals[k] - i_vals[k-1]) / dt
    else:  # centradas
        dt = t_vals[k+1] - t_vals[k-1]
        deriv = (i_vals[k+1] - i_vals[k-1]) / dt
    di_dt.append(deriv)

# Calcular e(t)
e_vals = [L * di_dt[k] + R * i_vals[k] for k in range(5)]

# Mostrar resultados
print(" t(s)   i(t) (A)   di/dt (A/s)   e(t) (V)")
for k in range(5):
    print(f"{t_vals[k]:.2f}   {i_vals[k]:.3f}      {di_dt[k]:.2f}         {e_vals[k]:.4f}")
```

✓ 0.0s

```
 t(s)   i(t) (A)   di/dt (A/s)   e(t) (V)
1.00   3.100      1.00         1.4202
1.01   3.110      1.00         1.4216
1.02   3.120      1.50         1.9130
1.03   3.140      3.00         3.3859
1.04   3.180      4.00         4.3716
```

3)

JACOBI

```python
import numpy as np

# Método de Jacobi
def metodo_jacobi(A, b, x_inicial, tol=1e-3, max_iter=100):
    print("Método de Jacobi")

    x_anterior = x_inicial.copy()  # Copia inicial del vector x
    n = len(b)  # Número de ecuaciones/variables

    for iteracion in range(1, max_iter + 1):
        x_nuevo = [0.0] * n  # Inicializa nuevo vector para esta iteración

        # Recorre cada ecuación
        for i in range(n):
            suma = 0
            # Calcula suma de términos A[i][j] * x[j] para j ≠ i
            for j in range(n):
                if i != j:
                    suma += A[i][j] * x_anterior[j]
            # Calcula nuevo valor de x[i]
            x_nuevo[i] = (b[i] - suma) / A[i][i]

        # Calcula errores absolutos para cada componente
        errores = [abs(x_nuevo[i] - x_anterior[i]) for i in range(n)]
        error = max(errores)  # Toma el mayor error (norma infinito)

        # Muestra resultados con 5 cifras significativas
        x_formato = [format(val, ".5g") for val in x_nuevo]
        print(f"Iteración {iteracion}: x = {x_formato}, error = {format(error, '.5g')}")

        if error < tol:  # Condición de parada si error es menor a la tolerancia
            break

        x_anterior = x_nuevo.copy()  # Actualiza x para la siguiente iteración

    return x_nuevo
```

Gauss-Seidel

```python
# Método de Gauss-Seidel
def metodo_gauss_seidel(A, b, x_inicial, tol=1e-3, max_iter=100):
    print("\nMétodo de Gauss-Seidel")

    x = x_inicial.copy()  # Vector de solución actual
    n = len(b)

    for iteracion in range(1, max_iter + 1):
        x_anterior = x.copy()  # Guarda valores anteriores para calcular el error

        # Recorre cada ecuación
        for i in range(n):
            suma = 0
            # Acumula suma de los otros términos A[i][j] * x[j] (usa valores ya actualizados)
            for j in range(n):
                if i != j:
                    suma += A[i][j] * x[j]
            # Actualiza inmediatamente el valor de x[i]
            x[i] = (b[i] - suma) / A[i][i]

        # Calcula el error como la mayor diferencia entre componentes
        errores = [abs(x[i] - x_anterior[i]) for i in range(n)]
        error = max(errores)

        # Muestra resultados con 5 cifras significativas
        x_formato = [format(val, ".5g") for val in x]
        print(f"Iteración {iteracion}: x = {x_formato}, error = {format(error, '.5g')}")

        if error < tol:  # Si el error es menor a la tolerancia, termina
            break

    return x

# Coeficientes del sistema
A = [
    [4, 2,  -3, 1],
    [1, 4, -1, -1],
    [-1, -2, 5, 1],
    [1, -2, 1, 3],
]
b = [-2, -1, 0, 1]
x0 = [0, 0, 0, 0]

sol_jacobi = metodo_jacobi(A, b, x0)

sol_seidel = metodo_gauss_seidel(A, b, x0)

print("\nSolución final Jacobi:       ", [format(val, ".5g") for val in sol_jacobi])
print("Solución final Gauss-Seidel: ", [format(val, ".5g") for val in sol_seidel])
```

```
        return x

    # Coeficientes del sistema
    A = [
        [4, 2,  -3, 1],
        [1, 4, -1, -1],
        [-1, -2, 5, 1],
        [1, -2, 1, 3],
    ]
    b = [-2, -1, 0, 1]
    x0 = [0, 0, 0, 0]

    sol_jacobi = metodo_jacobi(A, b, x0)

    sol_seidel = metodo_gauss_seidel(A, b, x0)

    print("\nSolución final Jacobi:       ", [format(val, ".5g") for val in sol_jacobi])
    print("Solución final Gauss-Seidel: ", [format(val, ".5g") for val in sol_seidel])
```

✓ 1.2s

```
Método de Jacobi
Iteración 1: x = ['-0.5', '-0.25', '0', '0.33333'], error = 0.5
Iteración 2: x = ['-0.45833', '-0.041667', '-0.26667', '0.33333'], error = 0.26667
Iteración 3: x = ['-0.7625', '-0.11875', '-0.175', '0.54722'], error = 0.30417
Iteración 4: x = ['-0.70868', '0.033681', '-0.30944', '0.56667'], error = 0.15243
Iteración 5: x = ['-0.89059', '-0.0085243', '-0.2416', '0.69516'], error = 0.18191
Iteración 6: x = ['-0.85073', '0.086039', '-0.32056', '0.70505'], error = 0.094563
Iteración 7: x = ['-0.9597', '0.058803', '-0.27674', '0.78112'], error = 0.10897
Iteración 8: x = ['-0.93224', '0.11602', '-0.32464', '0.78468'], error = 0.057218
Iteración 9: x = ['-0.99766', '0.098069', '-0.29698', '0.82964'], error = 0.065427
Iteración 10: x = ['-0.97918', '0.13258', '-0.32623', '0.83026'], error = 0.034513
Iteración 11: x = ['-1.0185', '0.1208', '-0.30885', '0.85686'], error = 0.039355
Iteración 12: x = ['-1.0063', '0.14163', '-0.32676', '0.85633'], error = 0.020833
Iteración 13: x = ['-1.03', '0.13396', '-0.31586', '0.87209'], error = 0.023712
Iteración 14: x = ['-1.0219', '0.14655', '-0.32683', '0.87125'], error = 0.012593
Iteración 15: x = ['-1.0362', '0.14158', '-0.32001', '0.88061'], error = 0.01431
Iteración 16: x = ['-1.0309', '0.1492', '-0.32673', '0.87979'], error = 0.0076227
Iteración 17: x = ['-1.0396', '0.146', '-0.32247', '0.88536'], error = 0.0086488
Iteración 18: x = ['-1.0362', '0.15062', '-0.32659', '0.88469'], error = 0.0046206
Iteración 19: x = ['-1.0414', '0.14857', '-0.32393', '0.88801'], error = 0.005235
Iteración 20: x = ['-1.0392', '0.15138', '-0.32646', '0.8875'], error = 0.0028046
Iteración 21: x = ['-1.0424', '0.15007', '-0.3248', '0.88948'], error = 0.0031731
Iteración 22: x = ['-1.041', '0.15177', '-0.32635', '0.88911'], error = 0.0017045
Iteración 23: x = ['-1.0429', '0.15094', '-0.32531', '0.8903'], error = 0.0019259
Iteración 24: x = ['-1.042', '0.15198', '-0.32627', '0.89004'], error = 0.0010371
Iteración 25: x = ['-1.0432', '0.15145', '-0.32562', '0.89075'], error = 0.0011704
Iteración 26: x = ['-1.0426', '0.15208', '-0.32621', '0.89058'], error = 0.00063179
```

```
Método de Gauss-Seidel
Iteración 1: x = ['-0.5', '-0.125', '-0.15', '0.46667'], error = 0.5
Iteración 2: x = ['-0.66667', '-0.0041667', '-0.22833', '0.62889'], error = 0.16667
Iteración 3: x = ['-0.82639', '0.056736', '-0.26836', '0.73607'], error = 0.15972
Iteración 4: x = ['-0.91366', '0.095343', '-0.29181', '0.79872'], error = 0.087269
Iteración 5: x = ['-0.96621', '0.11828', '-0.30567', '0.83615'], error = 0.05255
Iteración 6: x = ['-0.99743', '0.13198', '-0.31393', '0.85844'], error = 0.031224
Iteración 7: x = ['-1.016', '0.14014', '-0.31884', '0.87172'], error = 0.01861
Iteración 8: x = ['-1.0271', '0.145', '-0.32177', '0.87963'], error = 0.011088
Iteración 9: x = ['-1.0337', '0.1479', '-0.32351', '0.88435'], error = 0.0066072
Iteración 10: x = ['-1.0377', '0.14963', '-0.32455', '0.88716'], error = 0.003937
Iteración 11: x = ['-1.04', '0.15066', '-0.32517', '0.88884'], error = 0.002346
Iteración 12: x = ['-1.0414', '0.15127', '-0.32554', '0.88983'], error = 0.0013979
Iteración 13: x = ['-1.0422', '0.15164', '-0.32576', '0.89043'], error = 0.000833

Solución final Jacobi:        ['-1.0426', '0.15208', '-0.32621', '0.89058']
Solución final Gauss-Seidel:  ['-1.0422', '0.15164', '-0.32576', '0.89043']
```

4)

0

```python
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import inv, norm

# Paso a) Gráfica del sistema
x1_vals = np.linspace(-10, 10, 400)
x2_vals = np.linspace(-5, 15, 400)
X1, X2 = np.meshgrid(x1_vals, x2_vals)

# Funciones del sistema
F1 = -X1 * (X1 + 1) + 2 * X2 - 18
F2 = (X1 - 1) + (X2 - 6)**2 - 25

# Graficar curvas de nivel
plt.contour(X1, X2, F1, levels=[0], colors='blue', linewidths=2)
plt.contour(X1, X2, F2, levels=[0], colors='red', linewidths=2)
plt.xlabel("x1")
plt.ylabel("x2")
plt.title("Intersección de f1=0 y f2=0")
plt.grid(True)
plt.legend(["f1=0", "f2=0"])
plt.show()

# Aproximaciones visuales basadas en la gráfica
aprox_1 = np.array([-2.7, 11.4])
aprox_2 = np.array([1.5, 10.9])

# Paso b) Método de Newton-Raphson
def F(x):
    return np.array([
        -x[0] * (x[0] + 1) + 2 * x[1] - 18,
        (x[0] - 1) + (x[1] - 6)**2 - 25
    ])

def J(x):
    return np.array([
        [-2 * x[0] - 1, 2],
        [1, 2 * (x[1] - 6)]
    ])

def newton_raphson(x0, tol=1e-5, max_iter=50):
    x = x0.copy()
    print("Iteración 0: x1 = {:.6f}, x2 = {:.6f}".format(x[0], x[1]))
    for i in range(1, max_iter + 1):
        delta = inv(J(x)) @ F(x)
        x = x - delta
        norma_inf = norm(delta, ord=np.inf)
        print("Iteración {}: x1 = {:.6f}, x2 = {:.6f}, Norma inf = {:.2e}".format(i, x[0], x[1], norma_inf))
        if norma_inf < tol:
            break
    return x

print("\n--- Newton-Raphson desde Aproximación 1 ---")
sol1 = newton_raphson(aprox_1)

print("\n--- Newton-Raphson desde Aproximación 2 ---")
sol2 = newton_raphson(aprox_2)
```

5)

```python
import numpy as np

# Sistema de ecuaciones del problema
def F(x):
    x1, x2 = x
    return np.array([
        4 * x1**2 - 20 * x1 + (1/4) * x2**2 + 8,
        (1/2) * x1 * x2**2 + 2 * x1 - 5 * x2 + 8
    ])

# Jacobiana del sistema
def J(x):
    x1, x2 = x
    return np.array([
        [8 * x1 - 20, (1/2) * x2],
        [(1/2) * x2**2 + 2, x1 * x2 + (-5)]
    ])

# Aproximación inicial
x = np.array([0.0, 0.0])
print("Iteración 0: x1 = {:.4f}, x2 = {:.4f}".format(x[0], x[1]))

# Realizar 2 iteraciones del método de Newton-Raphson
for i in range(1, 4):
    inv_J = np.linalg.inv(J(x))
    x = x - inv_J.dot(F(x))
    print("Iteración {}: x1 = {:.4f}, x2 = {:.4f}".format(i, x[0], x[1]))
```

✓ 0.0s

```
Iteración 0: x1 = 0.0000, x2 = 0.0000
Iteración 1: x1 = 0.4000, x2 = 1.7600
Iteración 2: x1 = 0.4959, x2 = 1.9834
Iteración 3: x1 = 0.5000, x2 = 1.9999
```