Universidad Politécnica de Aguascalientes

Ingeniería en Sistemas Computacionales

Matemáticas para Ingeniería II

Parcial 3

Proyecto: Newton-Rhapson, Método de los trapecios.

Diego Emilio Rodríguez Rodríguez – UP230118

Santiago Alonso Silva Pedroza – UP230085

Kevin Enrique Avalos Rodríguez – UP230808

María Alejandra Farráez Reyes – UP230249

Edward Bejar Mateos - UP230430

ISC06A

Isaac Vázquez Mendoza

18/11/2025

main_launcher.py

Este programa sirve como lanzador (launcher) para abrir otros dos programas independientes:

- Metodo_Trapecios.py
- metodonewton.py

El usuario puede abrir cualquiera de los dos, o abrir ambos a la vez.

Manual de código:

Librerías utilizadas:

- os, sys: Permiten obtener rutas y datos del sistema operativo.
- subprocess: Permite abrir otros programas o scripts.
- tkinter: Usado para crear la interfaz gráfica.
- messagebox: Ventanas emergentes para errores.

(PY = sys.executable): Esta variable obtiene la ruta del intérprete Python que está ejecutando este launcher.

```
TRAP_FILE = "Metodo_Trapecios.py"
NEWTON_FILE = "metodonewton.py"
ECUA_FILE ="ecuacionewton.py"
```

run_script(filename):

Calcula la ruta absoluta del archivo, comprueba que el archivo realmente existe.

```
path = os.path.join(os.path.dirname(os.path.abspath(__file__)), filename)
if not os.path.exists(path):
    messagebox.showerror("Error", f"No se encontró {filename}\nRuta esperada: {path}")
    return
```

Si no existe: Muestra un error. Pero, si existe: Lo abre en un proceso separado usando:

```
subprocess.Popen([PY, path], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
```

No bloquea el launcher y cada script abre su propia ventana independiente.

```python
def abrir_trapecios():
    run_script(TRAP_FILE)

def abrir_newton():
    run_script(NEWTON_FILE)

def abrir_ecua():
    run_script(ECUA_FILE)
```

Funciones que abren los diferentes archivos.

crear_ventana():

Crea una interfaz.

```python
root = tk.Tk()
root.title("Launcher - Trapecios & Newton")
root.geometry("360x180")
tk.Label(root, text="Selecciona un programa para abrir:", font=("Segoe UI", 11)).pack(pady=10)

btn_frame = tk.Frame(root)
btn_frame.pack(pady=5)

tk.Button(btn_frame, text="Método del Trapecio", width=22, command=abrir_trapecios, bg="#b3e6b3").grid(row=0, column=0, padx=5,
tk.Button(btn_frame, text="Newton-Raphson", width=22, command=abrir_newton, bg="#cfe8ff").grid(row=1, column=0, padx=5, pady=5)
tk.Button(btn_frame, text="Newton-solucion_ecuaciones", width=22, command=abrir_ecua, bg="#ffd9b3").grid(row=2, column=0, padx=5

tk.Label(root, text="(Se abrirán en ventanas separadas)", font=("Segoe UI", 8)).pack(pady=6)
root.mainloop()
```

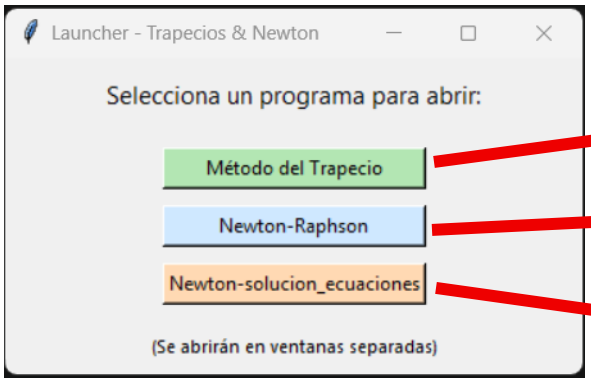
Hay tres botones:

1. Método del Trapecio
2. Newton-Raphson
3. Newton-solución ecuaciones

Cada botón ejecuta su función correspondiente.

El root.mainloop() mantiene la ventana viva.

Manual de uso:



1. Se ejecuta el archivo launcher.py.
2. Abrirá una ventana con tres botones.
3. Selecciona uno:
   a. Método del Trapecio: abre el programa para áreas numéricas.
   b. Newton-Raphson: abre el programa para resolver raíces.
   c. Newton-solución ecuaciones: abre el programa para encontrar dos raíces de una función.

Cada programa se abrirá en su propia ventana.

Si alguno de los archivos no existe, se mostrará un mensaje de error.

Metodo_Trapecios.py

Manual de código:

Librerías utilizadas:

- NumPy: Genera arreglos numéricos y manejar valores de x.
- Matplotlib: Grafica la función y los trapecios.
- Tkinter: Crea la ventana gráfica donde el usuario introduce los datos.
- Messagebox: Para mostrar errores emergentes.
- SymPy: Convierte la función ingresada por el usuario en una función matemática evaluable.

integral_trapecios():

Implementa el método del trapecio compuesto, dividiendo la integral en n trapecios.

1. Divide el intervalo (linspace): genera n+1 puntos entre *a* y *b*.
2. Evalúa la función: y_vals = f(x_vals).
3. Calcula el ancho de cada trapecio: h = (b - a) / n.

Fórmula del método del trapecio:

$$A \approx \frac{h}{2}\left[f(x_0) + 2\sum f(x_i) + f(x_n)\right]$$

=

(h / 2) * (y_vals[0] + 2 * sum(y_vals[1:-1]) + y_vals[-1])

Devuelve: área aproximada, valores de 'x', y valores de 'y'.

graficar():

Crea una figura grande para la gráfica:

```
plt.figure(figsize=(9, 6))
```

Dibuja la función suavizada: Se grafica con muchos puntos para que se vea bien.

```
x_smooth = np.linspace(a, b, 400)
plt.plot(x_smooth, f(x_smooth), label='f(x)', color='blue', linewidth=2)
```

Dibujo de trapecios: Cada trapecio se rellena en color rojo transparente.

Se dibujan como un polígono de 4 vértices:

- base izquierda
- punto arriba izquierdo
- punto arriba derecho
- base derecha

```
for i in range(len(x_vals)-1):
    xs = [x_vals[i], x_vals[i], x_vals[i+1], x_vals[i+1]]
    ys = [0, y_vals[i], y_vals[i+1], 0]
    plt.fill(xs, ys, color='red', alpha=0.4)
```

Mostrando el valor del área: Se coloca un recuadro en la esquina de la gráfica con el área calculada.

```
plt.text(
    0.05, 0.95,
    f"Área aproximada = {area:.6f}",
    transform=plt.gca().transAxes,
    fontsize=12,
    verticalalignment='top',
    bbox=dict(boxstyle="round", fc="white", ec="black")
)
```

transAxes: Coloca texto en coordenadas relativas del gráfico (0 a 1).

ejecutar():

Convierte el texto ingresado por el usuario en una función real de Python.

Ejemplo: Escribes: *ejemplo de función*, sympy lo convierte en una expresión simbólica, y lambdify lo convierte en una función numpy que se puede evaluar.

```
x = symbols('x')
f_expr = sympify(entry_funcion.get())
f = lambdify(x, f_expr, 'numpy')
```

Obtiene valores de la interfaz gráfica: Toma los límites y número de trapecios que escribió el usuario.

```
a = float(entry_a.get())
b = float(entry_b.get())
n = int(entry_n.get())
```

Calcula el área y manda a graficar: Muestra el resultado en pantalla (caja de texto) y luego genera la gráfica.

```
area, x_vals, y_vals = integral_trapecios(f, a, b, n)

text_output.delete("1.0", tk.END)
text_output.insert(tk.END, f"Área aproximada: {area:.6f}\n")

graficar(f, a, b, x_vals, y_vals, area)
```

Si ocurre un error salta el siguiente exception:

```
except Exception as e:
    messagebox.showerror("Error", f"Ocurrió un error: {str(e)}")
```

Interfaz gráfica en Tkinter:

```
ventana = tk.Tk()
ventana.title("Integral por método del trapecio")
ventana.geometry("600x550")

tk.Label(ventana, text="Función f(x):").pack()
entry_funcion = tk.Entry(ventana, width=40)
entry_funcion.insert(0, "sin(x)")
entry_funcion.pack()

tk.Label(ventana, text="Límite inferior a:").pack()
entry_a = tk.Entry(ventana)
entry_a.insert(0, "0")
entry_a.pack()

tk.Label(ventana, text="Límite superior b:").pack()
entry_b = tk.Entry(ventana)
entry_b.insert(0, "3.1416")
entry_b.pack()

tk.Label(ventana, text="Número de trapecios:").pack()
entry_n = tk.Entry(ventana)
entry_n.insert(0, "10")
entry_n.pack()

tk.Button(ventana, text="Calcular integral", command=ejecutar, bg="lightgreen").pack(pady=10)

tk.Label(ventana, text="Resultados:").pack()
text_output = tk.Text(ventana, height=10, width=70)
text_output.pack()
```

tk.Label (*donde va a estar la etiqueta* (ventana), *texto que tendrá* (text =)).pack()

= tk.Entry(ventana): Lo hace input.

.insert(0,”0”): Le da un valor dentro por default.

tk.Text (*donde va a estar* (ventana), *altura*, *anchura*)

ventana: Crear la ventana principal, donde todos los inputs y outputs se colocan.

entry_funcion: Campo donde el usuario escribe f(x).

entry_a: Límite inferior.

entry_b: Límite superior.

entry_n: Número de trapecios.

Button: Acciona todo el código (ejecutar()).

text_output: Área aproximada en texto.

La ventana usa “.pack()” para colocar todo verticalmente.

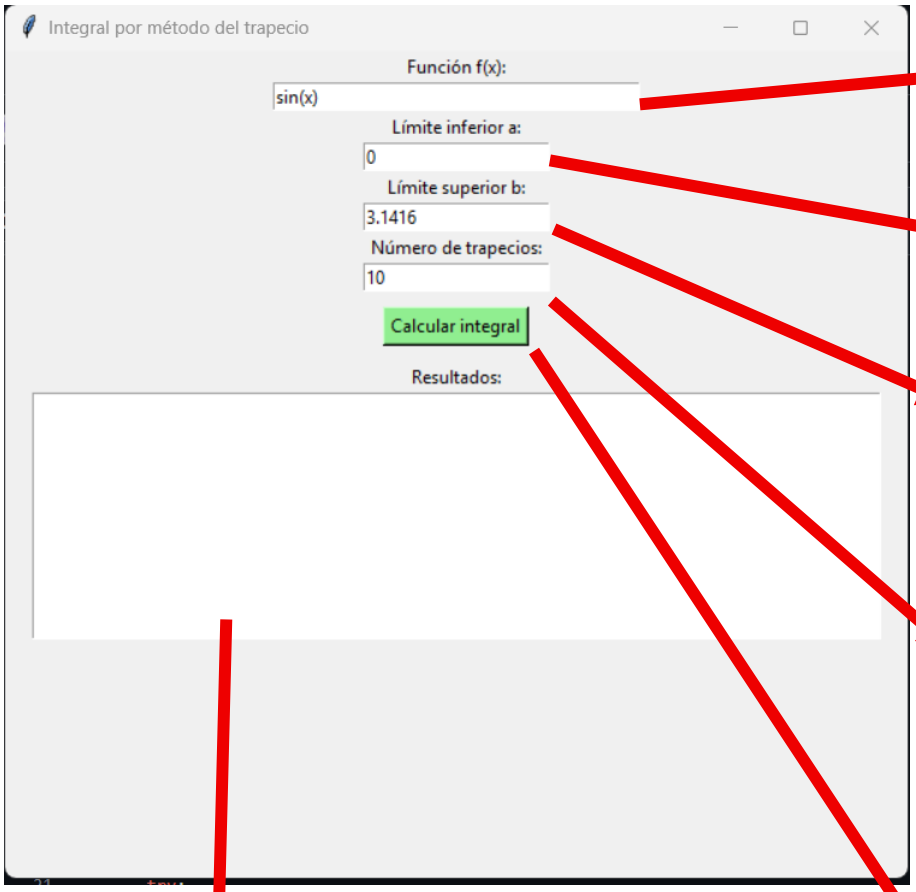
mainloop(): Es el bucle infinito que mantiene la ventana abierta.

Manual de uso:

Tras ejecutar el archivo .py, aparecerá una ventana llamada: “Integral por método del trapecio”

Esta ventana contiene:

- Entradas para definir función y límites
- Botón para calcular
- Área para mostrar resultados numéricos



entry_function: Escribir la función que se desea integrar. La variable siempre debe llamarse “x”

entry_a: Aquí escribe el valor inicial del intervalo de integración.

entry_b: Aquí escribe el valor final del intervalo. Debe ser mayor que “a”.

entry_n: Aquí escribe cuántos trapecios usará el método para aproximar el área.

Debe ser un número entero positivo.

text_output: Los resultados se muestran en la sección “Resultados:”

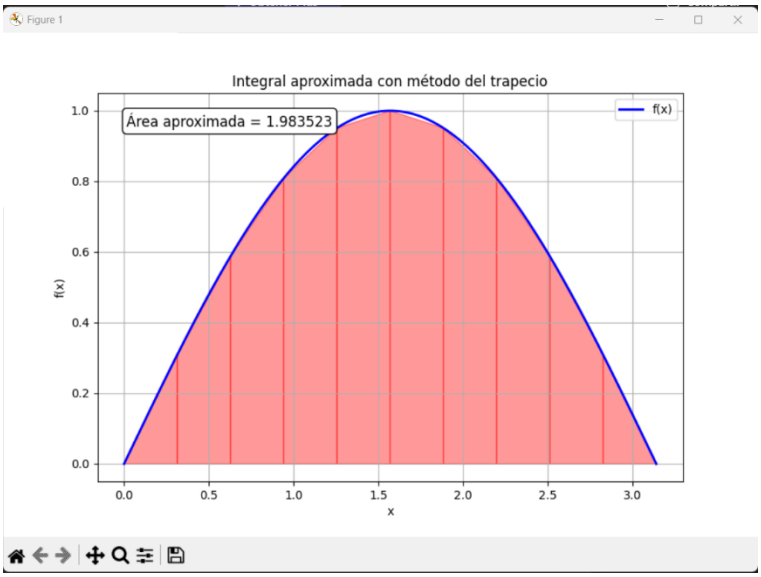
button:

- Se procesa la función ingresada.
- Se calcula el área usando el método del trapecio.
- Se imprimen los resultados numéricos en la caja inferior.
- Se abre una gráfica.

Al finalizar el cálculo aparece una gráfica con:

- La curva de f(x) (línea azul).
- Los trapecios (áreas rojas)
- Eje ‘X’ y ‘Y‘ marcados.
- Cuadro en la esquina superior mostrando el área aproximada.

Esta representación permite visualizar cómo se aproxima la integral mediante figuras geométricas.



Mensajes de error comunes:

Problema:	Causa:
“Función inválida”	La función está mal escrita.
“No se puede convertir a número”	‘a’, ‘b’ o ‘n’ no son valores numéricos.
“n debe ser positivo”	Se ingresó valor negativo o cero.
Gráfica vacía	func = constante 0.

metodonewton.py

Este programa permite aproximar raíces de funciones utilizando el método de Newton-Raphson con una interfaz gráfica amigable y fácil de usar, el usuario puede ingresar la función, su derivada y un valor inicial y este genera una gráfica ilustrando la convergencia.

El programa calcula cada iteración, muestra los resultados en pantalla y genera una gráfica que incluye:

- La función
- Su derivada
- Los puntos generados por cada iteración
- La raíz aproximada

Manual de código:

1. Librerías: Tkinter, numpy, matplotlib.pyplot

2. Función principal: Es la encargada de ejecutar el método de Newton-Raphson y de controlar su proceso.

   2.1. Lectura de entradas del usuario:

```
# Leer entradas del usuario
funcion_str = entry_funcion.get()
derivada_str = entry_derivada.get()
x0 = float(entry_x0.get())
iter_user = int(entry_iter.get())  # Iteraciones que solicita el usuario
```

   2.2. Definición de la función y derivada: Se utiliza eval() para convertir el texto ingresado en una función ejecutable.

```
def f(x):
    return eval(funcion_str, {"x": x, "np": np,
                              "sin": np.sin, "cos": np.cos, "exp": np.exp,
                              "log": np.log, "sqrt": np.sqrt})
```

   2.3. Ciclo iterativo del método: Cada iteración se almacena en una lista y se imprime en la pantalla.

```
x1 = x0 - fx / dfx
iteraciones.append(x1)
```

2.4. Condición de detención: Se detendrá si la función se acerca suficientemente a cero.

```
# Si la función llega cerca de 0, detener
if abs(f(x1)) < 1e-6:
    convergio_en = i + 1
    texto_resultados.insert(tk.END, "\nRaíz encontrada antes de las iteraciones solicitadas.\n")
    break
```

2.5. Manejo de errores: Se mostrará un mensaje de “Error, la derivada es cero, no se puede continuar.”

```
if dfx == 0:
    messagebox.showerror("Error", "La derivada es cero. No se puede continuar.")
    return
```

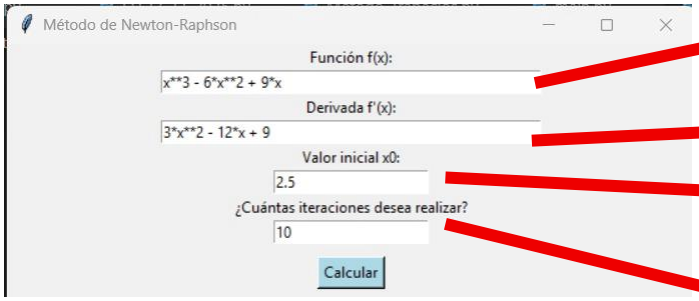
3. Generación de graficas: Al finalizar el cálculo, el programa grafica la función, la derivada, marca los puntos de iteración, señala la raíz encontrada.

```
plt.plot(x_vals, y_vals, label=f'f(x) = {funcion_str}', color='blue')
plt.plot(x_vals, y_deriv, label="f'(x)", color='orange', linestyle='--')
plt.scatter(iteraciones, [f(x) for x in iteraciones], color='red', label='Iteraciones')
```

4. Interfaz gráfica: la ventana principal contiene función, derivada, valor inicial, número de iteraciones.

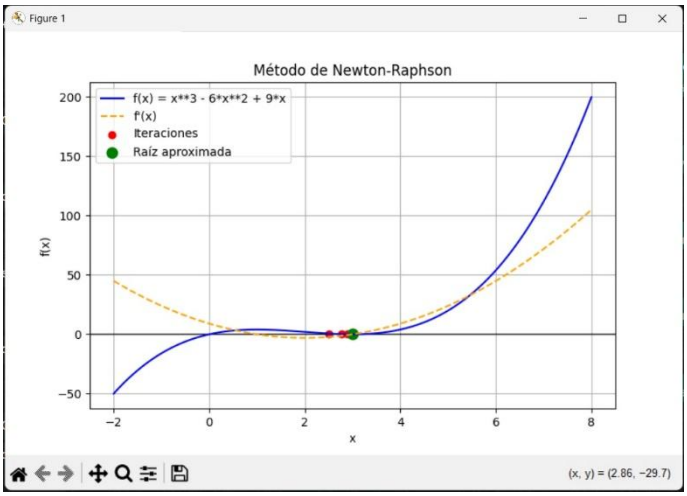
```
116     tk.Button(ventana, text="Calcular", command=newton_raphson, bg="lightblue").pack(pady=10)
117
```

Manual de uso:



- Primer campo función: Se ingresa la función matemática
- Segundo campo derivada: Se ingresa la derivada correspondiente
- Tercer campo valor inicial: Se ingresa el punto donde iniciara el método
- Cuarto campo iteraciones: Se ingresa el número de iteraciones que se desea realizar.

Área de resultados: Aparece la gráfica que contiene la función en línea azul, la derivada que es la línea naranja discontinua, punto rojo que simbolizan las iteraciones, punto verde que marca la raíz aproximada.




```
Iteraciones del método de Newton-Raphson:

Iteración 1: x = 2.777778, f(x) = 0.625000, f'(x) = -2.250000
Iteración 2: x = 2.893519, f(x) = 0.137174, f'(x) = -1.185185
Iteración 3: x = 2.947757, f(x) = 0.032808, f'(x) = -0.604874
Iteración 4: x = 2.974112, f(x) = 0.008045, f'(x) = -0.305269
Iteración 5: x = 2.987113, f(x) = 0.001993, f'(x) = -0.153316
Iteración 6: x = 2.993570, f(x) = 0.000496, f'(x) = -0.076826
Iteración 7: x = 2.996789, f(x) = 0.000124, f'(x) = -0.038454
Iteración 8: x = 2.998395, f(x) = 0.000031, f'(x) = -0.019238
Iteración 9: x = 2.999198, f(x) = 0.000008, f'(x) = -0.009621
Iteración 10: x = 2.999599, f(x) = 0.000002, f'(x) = -0.004811

Raíz encontrada antes de las iteraciones solicitadas.
```

Y te muestra el número de iteraciones (con su x, función de x, y la derivada de dicha función), dependiendo de lo ingresado, y avisa si se encontró la raíz en una iteración previa a la ingresada.

ecuacionewton.py

Este programa implementa:

- El método de Newton-Raphson para encontrar dos raíces de una función.
- La reconstrucción de la ecuación diferencial lineal asociada.
- Gráficas:
    - De la función característica.
    - De las iteraciones del método de Newton.
    - De la solución $x(t)$.

Manual de código:

Librerías utilizadas:

- tkinter: crear la interfaz gráfica.
- numpy: manejar arreglos numéricos.
- matplotlib: generar gráficas.
- sympy: manipular expresiones matemáticas y obtener coeficientes.

newton_all_steps():

```python
def newton_all_steps(func_str, deriv_str, x0, iters):
    """Devuelve la raíz y todas las iteraciones para graficar."""
    def f(x):
        return eval(func_str, {"x": x, "np": np})

    def fprima(x):
        return eval(deriv_str, {"x": x, "np": np})

    xs = [x0]

    for _ in range(iters):
        fx = f(x0)
        dfx = fprima(x0)

        if dfx == 0:
            raise ValueError("La derivada se hizo cero. No se puede continuar.")

        x1 = x0 - fx / dfx
        xs.append(x1)

        if abs(f(x1)) < 1e-8:
            return x1, xs

        x0 = x1

    return x1, xs
```

- func_str: La función f(x).
- deriv_str: La derivada f'(x).
- x0: Valor inicial.
- iters: Número de iteraciones.

Convierte los textos (inputs) en funciones ejecutables usando eval.

Guarda la secuencia de valores que Newton va generando.

Aplica la fórmula:

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

=

x1 = x0 – fx / dfx

Se detiene si la derivada es cero (**error**), o si la función se acerca a cero (**raíz encontrada**).

Devuelve: La raíz encontrada, La lista completa de iteraciones (para graficar).

calcular_todo():

```
funcion_str = entry_funcion.get()
derivada_str = entry_derivada.get()
x0_1 = float(entry_x0_1.get())
x0_2 = float(entry_x0_2.get())
iter_user = int(entry_iter.get())
```

Obtiene los datos desde la interfaz.

- Función.
- Derivada.
- Dos valores iniciales.
- Iteraciones

Limpia el cuadro de texto y muestra el título.

```
texto_resultados.delete(1.0, tk.END)
texto_resultados.insert(tk.END, "=== MÉTODO DE NEWTON PARA LAS DOS RAÍCES ===\n\n")
```

Obtención automática de coeficientes a, b, y c.

```
x = sp.Symbol("x")
pol = sp.sympify(funcion_str)
pol = sp.expand(pol)

a = float(pol.coeff(x, 2))
b = float(pol.coeff(x, 1))
c = float(pol.coeff(x, 0))
```

Sympy convierte la función en polinomio e identifica los coeficientes, para reconstruir la ecuación diferencial.

ax'' + bx' + cx = 0

Aplica Newton para obtener las dos raíces, m1 es la primera raíz, m2 es la segunda, its1 y its2 son las listas de iteraciones.

```
m1, its1 = newton_all_steps(funcion_str, derivada_str, x0_1, iter_user)
texto_resultados.insert(tk.END, f"Raíz 1 (m1) usando x0 = {x0_1} → {m1}\n")

# Newton para m2
m2, its2 = newton_all_steps(funcion_str, derivada_str, x0_2, iter_user)
texto_resultados.insert(tk.END, f"Raíz 2 (m2) usando x0 = {x0_2} → {m2}\n\n")
```

Determina si las raíces son iguales o distintas. Si son casi iguales es una raíz doble.

```
if abs(m1 - m2) < 1e-6:
    caso = "Caso I (Raíz doble)"
    texto_resultados.insert(tk.END, "-> CASO I: Raíz doble\n")
    texto_resultados.insert(
        tk.END,
        f"x(t) = k1 · e^({m1:.6f}·t)  +  t · k2 · e^({m1:.6f}·t)\n"
    )
else:
    caso = "Caso II (Raíces distintas)"
    texto_resultados.insert(tk.END, "-> CASO II: Raíces distintas\n")
    texto_resultados.insert(
        tk.END,
        f"x(t) = k1 · e^({m1:.6f}·t)  +  k2 · e^({m2:.6f}·t)\n"
    )
```

Caso 1: Raíz Doble.

$$x(t) = k_1 e^{mt} + t k_2 e^{mt}$$

Caso 2: Raíces Distintas.

$$x(t) = k_1 e^{m_1 t} + k_2 e^{m_2 t}$$

Muestra la ecuación diferencial original.

```
texto_resultados.insert(tk.END, f"{a}x'' + {b}x' + {c}x = 0\n\n")
```

```
xs = np.linspace(-10, 10, 400)
f_vals = [eval(funcion_str, {"x": x, "np": np}) for x in xs]

plt.figure(figsize=(8,5))
plt.axhline(0, color="black")
plt.plot(xs, f_vals, label="f(m)")
plt.scatter([m1, m2], [0, 0], color="red", s=80, label="Raíces encontradas")
plt.title("Función Característica")
plt.xlabel("m")
plt.ylabel("f(m)")
plt.grid(True)
plt.legend()
plt.show()

# Gráfica de iteraciones de Newton para m1
plt.figure(figsize=(8,5))
its_f = [eval(funcion_str, {"x": x, "np": np}) for x in its1]
plt.plot(its1, its_f, marker="o")
plt.axhline(0, color="black")
plt.title("Iteraciones Newton - Raíz m1")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(True)
plt.show()

# Gráfica de iteraciones de Newton para m2
plt.figure(figsize=(8,5))
its_f2 = [eval(funcion_str, {"x": x, "np": np}) for x in its2]
plt.plot(its2, its_f2, marker="o")
plt.axhline(0, color="black")
plt.title("Iteraciones Newton - Raíz m2")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(True)
```

1) Gráfica de la función característica

Muestra el polinomio f(m) y marca las raíces encontradas.

2) Iteraciones de Newton para cada raíz

Se dibuja cómo Newton va bajando hacia la raíz en cada paso.

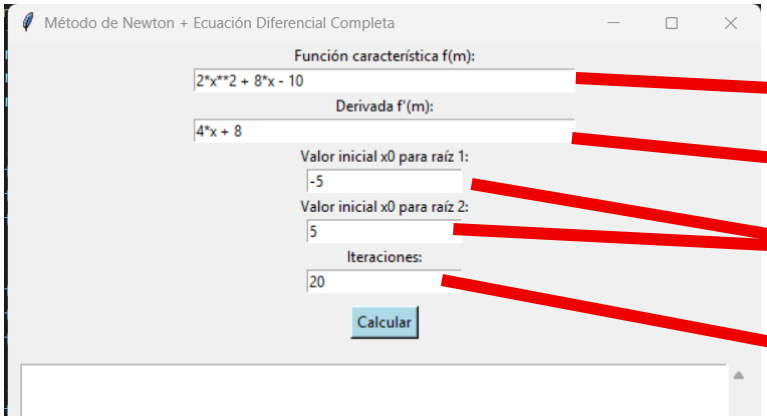
3) Gráfica de la solución diferencial x(t)

Dependiendo del caso:

- Raíz doble
- Raíces distintas

Se grafica la solución correspondiente.

Manual de Usuario:



Aquí metes la función característica.

Aquí metes la derivada.

Aquí metes los valores iniciales para las raíces.

Numero de iteraciones.

```
Ecuación característica:
 2.0m² + 8.0m + -10.0 = 0

Raíz 1 (m1) usando x0 = -5.0 → -5.0
Raíz 2 (m2) usando x0 = 5.0 → 1.0000000000001107

-> CASO II: Raíces distintas
x(t) = k1 · e^(-5.000000·t)  +  k2 · e^(1.000000·t)

=== ECUACIÓN DIFERENCIAL ORIGINAL ===
2.0x'' + 8.0x' + -10.0x = 0
```

Ecuación característica.

Aplica Newton dos veces para los valores de las raíces.

Te dice el tipo de caso, y la solución de x(t).

Ecuación original.

Saldrá una gráfica, una vez cierres una saldrá otras consecutivamente.