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

01/12/2025

Índice:

main_launcher.py

Este programa sirve como lanzador (launcher) para abrir otros seis programas independientes.

El usuario puede abrir cualquiera de los programas.

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"
EULER_FILE = "EULER.py"
EULER_UP_FILE = "eulermejorado.py"
RK4_FILE = "RK4.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)

def abrir_euler():
    run_script(EULER_FILE)

def abrir_euler_up():
    run_script(EULER_UP_FILE)

def abrir_rk4():
    run_script(RK4_FILE)
```

Funciones que abren los diferentes archivos.

crear_ventana():

Crea una interfaz.

```python
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=0, column=1, padx=5, pady=5)
tk.Button(btn_frame, text="Newton Solución Ecuaciones", width=22, command=abrir_ecua, bg="#ffd9b3").grid(row=1, column=0, padx=5
tk.Button(btn_frame, text="Método de Euler", width=22, command=abrir_euler, bg="#ebfe6e").grid(row=1, column=1, padx=5, pady=5)
tk.Button(btn_frame, text="Euler Mejorado", width=22, command=abrir_euler_up, bg="#e5aaef").grid(row=2, column=0, padx=5, pady=5
tk.Button(btn_frame, text="Método RK4", width=22, command=abrir_rk4, bg="#f67878").grid(row=2, column=1, padx=5, pady=5)
```

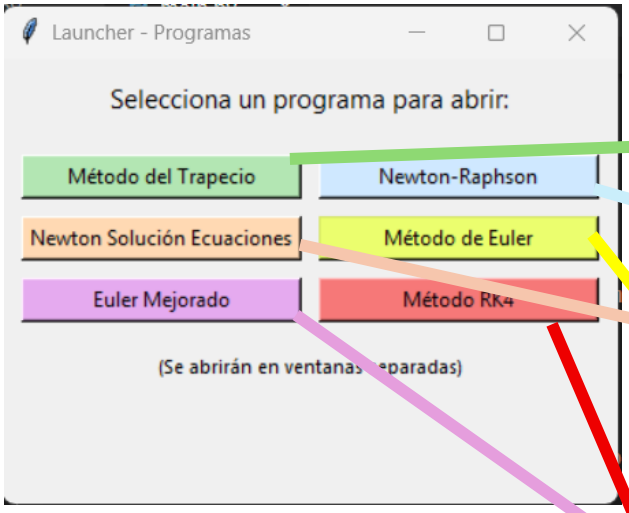
Hay seis botones:

1. Método del Trapecio.
2. Newton-Raphson.
3. Newton Solución Ecuaciones.
4. Método de Euler.
5. Euler Mejorado.
6. Método RK4.

Cada botón ejecuta su función correspondiente.

El root.mainloop() mantiene la ventana viva.

Manual de Usuario:



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.
   d. Método de Euler: abre el programa para resolver numéricamente una ecuación diferencial ordinaria de la forma.
   e. Euler Mejorado: abre el programa para que resuelva ecuaciones diferenciales ordinarias de primer orden de la forma usando el Método de Euler Mejorado.
   f. Método RK4: abre el programa para resolver numéricamente ecuaciones diferenciales de primer orden usando el método Runge-Kutta clásico de 4to orden.

Cada programa se abrirá en su propia ventana.

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

Pseudocódigo:

INICIAR programa

OBTENER la ruta del intérprete de Python actual

DEFINIR los nombres de los archivos:

- archivo del método del trapecio
- archivo de Newton-Raphson
- archivo de Newton para sistemas
- archivo del método de Euler
- archivo de Euler mejorado
- archivo del método RK4

```
FUNCIÓN run_script():
    Construir la ruta completa del archivo usando el directorio actual y filename
    SI el archivo NO existe
        Mostrar mensaje de error indicando que no se encontró filename
        TERMINAR FUNCIÓN
    INTENTAR
        Lanzar el archivo Python en un proceso nuevo usando:
            subprocess.Popen([PY, path])
    SI ocurre algún error
        Mostrar mensaje de error con detalles del fallo
```

```
FUNCIÓN abrir_trapecios()
    Llamar run_script(TRAP_FILE)
FUNCIÓN abrir_newton():
    Llamar run_script(NEWTON_FILE)
FUNCIÓN abrir_ecua():
    Llamar run_script(ECUA_FILE)
FUNCIÓN abrir_euler():
    llamar a run_script(archivo_euler)
FUNCIÓN abrir_euler_up():
    llamar a run_script(archivo_euler_mejorado)
FUNCIÓN abrir_rk4():
    llamar a run_script(archivo_rk4)
```

```
FUNCIÓN crear_ventana():
    crear ventana principal
    colocar título y tamaño
    mostrar texto: "Selecciona un programa para abrir"
    crear contenedor para los botones
    CREAR botón "Método del Trapecio" -> abrir_trapecios
    CREAR botón "Newton-Raphson" -> abrir_newton
    CREAR botón "Newton Solución Ecuaciones" -> abrir_ecua
    CREAR botón "Método de Euler" -> abrir_euler
    CREAR botón "Euler Mejorado" -> abrir_euler_up
    CREAR botón "Método RK4" -> abrir_rk4
    mostrar texto de aclaración "(Se abrirán en ventanas separadas)"
    iniciar el ciclo principal de la interfaz
SI este archivo es el programa principal:
    llamar a crear_ventana()
TERMINAR programa
```

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.

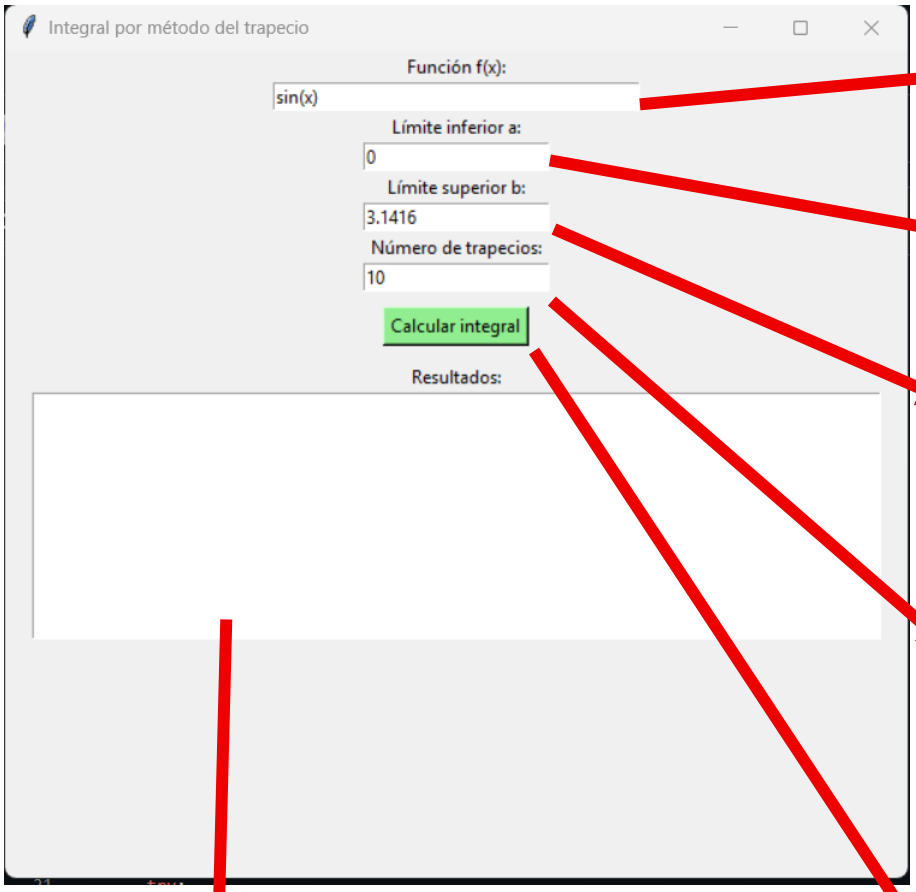
mainloop(): Es el bucle infinito que mantiene la ventana abierta.

Manual de Usuario:

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:”

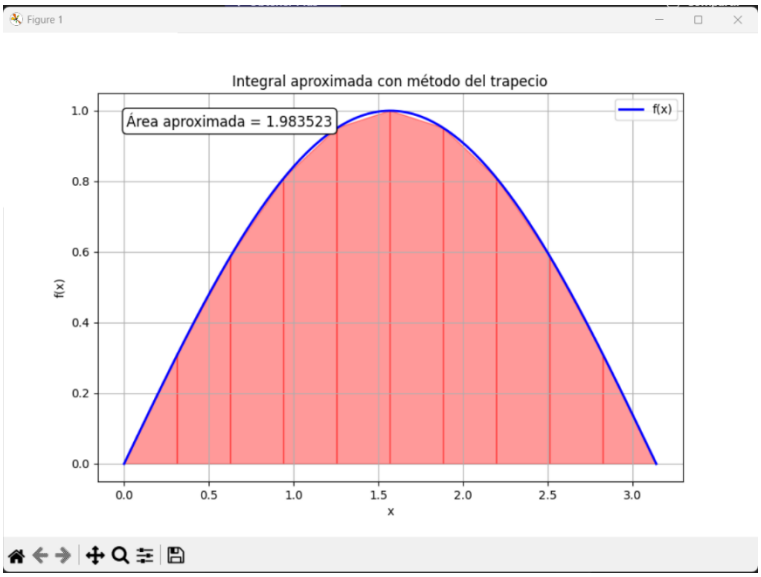
button:

- 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.

Pseudocódigo:

FUNCIÓN integral_trapecios():

Crear un arreglo x_vals con n+1 puntos equidistantes entre a y b

Crear un arreglo y_vals con la evaluación de f en cada punto de x_vals

h ← (b - a) / n // tamaño del subintervalo

area ← (h / 2) * (y_vals[0]

+ 2 * suma de los valores y_vals desde el índice 1 hasta n-1

+ y_vals[n])

REGRESAR (area, x_vals, y_vals)

FUNCIÓN graficar():

Crear una figura para graficar

Crear x_smooth con 400 puntos entre a y b

Graficar la curva suave de f(x_smooth)

PARA i desde 0 hasta longitud(x_vals) - 2 HACER

Crear xs = [x_i, x_i, x_(i+1), x_(i+1)]

Crear ys = [0, y_i, y_(i+1), 0]

Dibujar el trapecio usando xs y ys en color rojo

FIN PARA

Mostrar en la gráfica el texto: "Área aproximada = area"

Agregar título, etiquetas y cuadrícula

Mostrar la gráfica

FUNCIÓN ejecutar():

INTENTAR

Crear símbolo x para SymPy

Leer texto de entrada y convertirlo a expresión matemática f_expr

Convertir f_expr a una función evaluable f

Leer valores a, b, n desde los campos de entrada

(area, x_vals, y_vals) ← integral_trapecios(f, a, b, n)

Limpiar el cuadro de texto de resultados

Escribir "Área aproximada: area" en el cuadro de texto

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

SI OCURRE ERROR

Mostrar ventana emergente con el mensaje de error

INTERFAZ TKINTER:

Crear ventana principal

Configurar título y tamaño

Crear etiqueta "Función f(x)"

Crear campo de texto para función con valor por defecto "sin(x)"

Crear etiqueta "Límite inferior"

Crear campo entry_a con valor "0"

Crear etiqueta "Límite superior"

Crear campo entry_b con valor "3.1416"

Crear etiqueta "Número de trapecios"

Crear campo entry_n con valor "10"

Crear botón "Calcular integral" que llama a ejecutar()

Crear etiqueta "Resultados"

Crear cuadro de texto para mostrar resultados

Iniciar el ciclo principal de la ventana

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
```

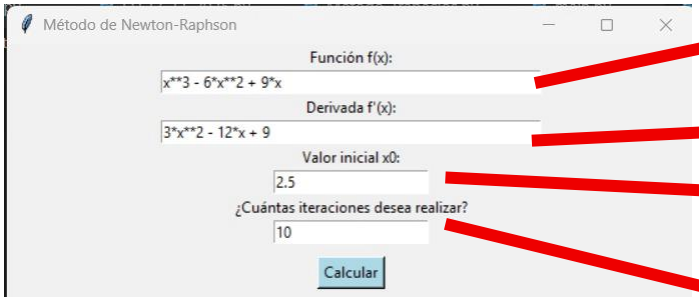
3. Generación de graficas: Al finalizar el cálculo, el programa grafica la función, la derivada, marca los puntos de iteración, señala la raíz encontrada.

```
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.

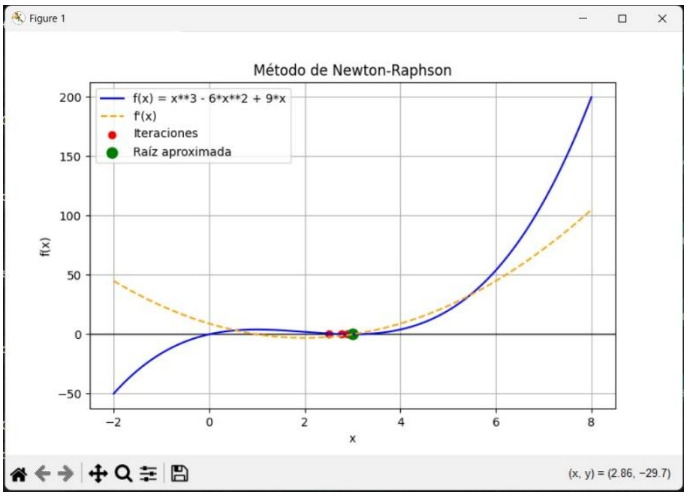
```
116     tk.Button(ventana, text="Calcular", command=newton_raphson, bg="lightblue").pack(pady=10)
117
```

Manual de Usuario:



- 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.

Pseudocódigo:

```
FUNCIÓN newton_raphson():
    INTENTAR
        Leer la función ingresada como texto (funcion_str)
        Leer la derivada ingresada como texto (derivada_str)
        Leer valor inicial x0
        Leer número de iteraciones solicitadas
        Definir función f(x):
          Evaluar funcion_str usando eval con funciones matemáticas permitidas
        Definir función fprima(x):
          Evaluar derivada_str usando eval con funciones matemáticas permitidas
        Limpiar el cuadro de texto donde se mostrarán resultados
        Escribir encabezado "Iteraciones del método..."
        Crear lista iteraciones con el valor inicial x0
        Definir convergio_en = NULO
        PARA i desde 0 hasta iter_user - 1 HACER
          Calcular fx = f(x0)
          Calcular dfx = fprima(x0)
          SI la derivada es cero
            Mostrar error "No se puede continuar"
            TERMINAR FUNCIÓN
          Calcular x1 = x0 - fx / dfx
          Agregar x1 a lista iteraciones
          Mostrar en texto:
            Número de iteración, valor de x1, f(x0), f'(x0)
          SI |f(x1)| < 1e-6
            convergio_en = i + 1
            Mostrar "Raíz encontrada antes de las iteraciones"
            SALIR del bucle
          Actualizar x0 = x1
        FIN PARA
        SI convergio_en existe Y convergio_en < iter_user
          Mostrar que se necesitaron menos iteraciones
        SINO
          Mostrar que se hicieron todas las iteraciones solicitadas
        Mostrar en pantalla "Raíz aproximada: x1"
```

```
        GRAFICAR:
            Crear un rango de valores x_vals alrededor de x1
            Calcular y_vals = f(x) para cada punto
            Calcular y_deriv = f'(x) para cada punto
            Crear figura
            Dibujar eje horizontal en 0
            Dibujar curva de f(x)
            Dibujar curva de f'(x) con línea punteada
            Dibujar puntos de las iteraciones
            Dibujar punto especial para la raíz aproximada
            Agregar título, etiquetas, cuadrícula y leyenda
            Mostrar gráfica
        SI ocurre cualquier error
            Mostrar ventana de error
```

```
INTERFAZ GRÁFICA ():
        Crear ventana principal
          Establecer título "Método de Newton-Raphson"
          Establecer tamaño 550x600
          Crear etiqueta "Función f(x)"
          Crear campo de texto entry_funcion
          Insertar ejemplo por defecto
          Crear etiqueta "Derivada f'(x)"
          Crear campo entry_derivada
          Insertar ejemplo por defecto
          Crear etiqueta "Valor inicial x0"
          Crear campo entry_x0
          Insertar valor por defecto
          Crear etiqueta "¿Cuántas iteraciones desea realizar?"
          Crear campo entry_iter
          Insertar valor por defecto
          Crear botón "Calcular"
            Al presionarlo ejecuta newton_raphson()
          Crear cuadro de texto con scroll (scrolledtext) para mostrar resultados
          Iniciar mainloop de Tkinter
```

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.

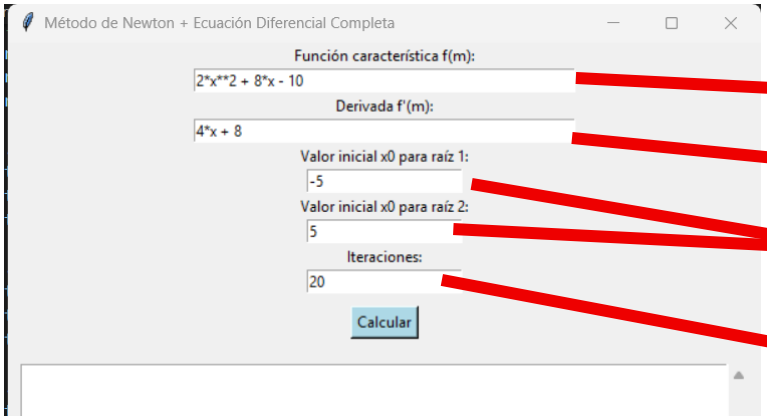
3) Gráfica de la solución diferencial x(t)

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
```

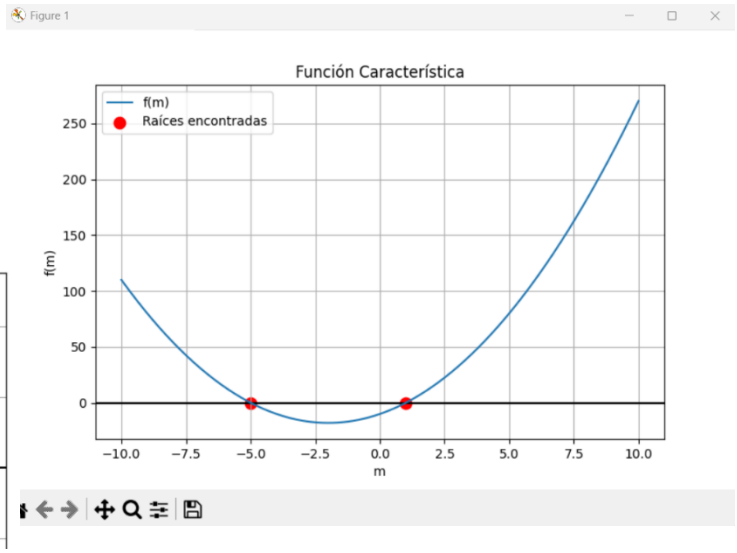
Ecuación característica.

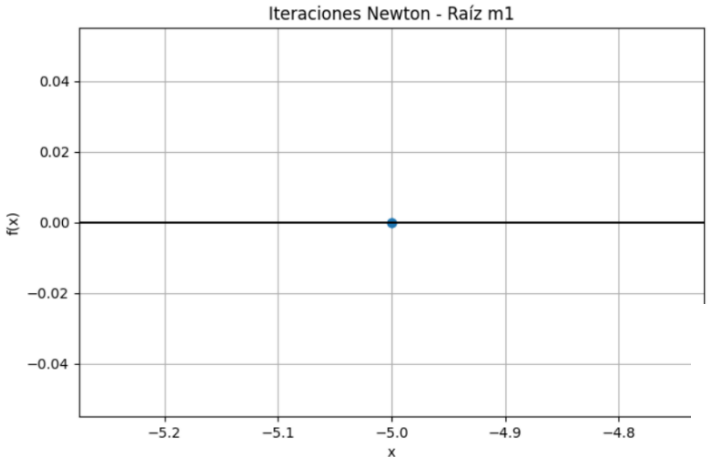
Aplica Newton dos veces para los valores de las raíces.

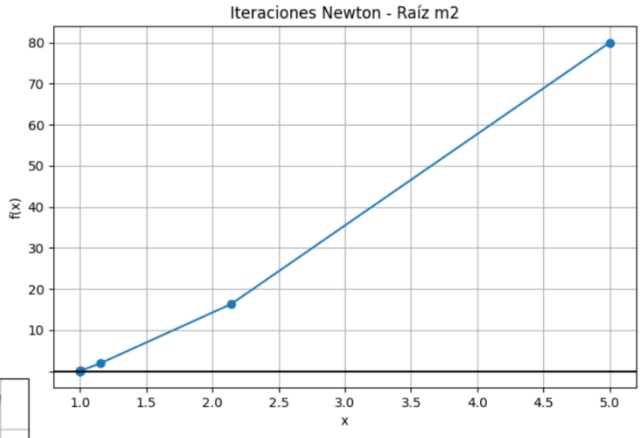
Te dice el tipo de caso, y la solución de x(t).

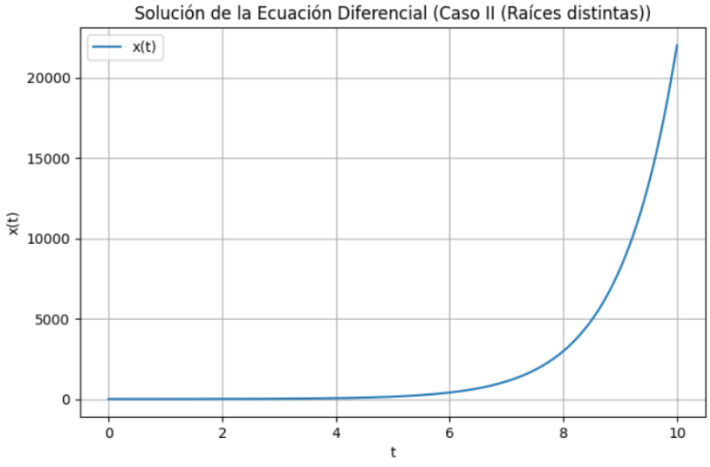
Ecuación original.

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









Pseudocódigo:

FUNCIÓN calcular_todo():

INTENTAR

Leer textos:

funcion_str

derivada_str

x0_1

x0_2

iter_user

Limpiar zona de resultados

Escribir encabezado

--- EXTRAER A, B, C ---

Definir variable simbólica x

Convertir funcion_str a expresión SymPy

Expandir el polinomio

Obtener coeficientes:

a = coeficiente de x^2

b = coeficiente de x^1

c = coeficiente de x^0

Mostrar la ecuación característica: a m^2 + b m + c = 0

--- NEWTON PARA M1 ---

Llamar newton_all_steps(func_str, deriv_str, x0_1, iter_user)

Guardar raíz m1 y lista its1

Mostrar m1

--- NEWTON PARA M2 ---

Llamar newton_all_steps(func_str, deriv_str, x0_2, iter_user)

Guardar raíz m2 y lista its2

Mostrar m2

--- CLASIFICACIÓN DEL SISTEMA DIFERENCIAL ---

SI |m1 - m2| < tolerancia

CASO: Raíz doble

FUNCIÓN newton_all_steps():

Definir función f(x):

Evaluar func_str usando eval

Definir función fprima(x):

Evaluar deriv_str usando eval

Crear lista xs con el valor inicial x0

PARA cada iteración desde 1 hasta iters HACER

Calcular fx = f(x0)

Calcular dfx = fprima(x0)

SI dfx = 0

Lanzar error: "La derivada se hizo cero"

Calcular x1 = x0 - fx / dfx

Agregar x1 a xs

SI |f(x1)| < 1e-8

REGRESAR (x1, xs)

Actualizar x0 = x1

FIN PARA

REGRESAR (x1, xs), aunque no haya convergido antes de tiempo

INTERFAZ GRÁFICA:

Crear ventana principal

Configurar título y tamaño

Crear etiqueta y entrada para función característica

Colocar valor por defecto

Crear etiqueta y entrada para derivada

Colocar valor por defecto

Crear etiqueta y entrada para x0 de raíz 1

Crear etiqueta y entrada para x0 de raíz 2

Crear etiqueta y entrada para iteraciones

Crear botón "Calcular" que ejecuta calcular_todo()

Crear cuadro de texto con scroll para mostrar resultados

Iniciar ventana principal

Mostrar:

x(t) = k1 e^(m t) + t k2 e^(m t)

SINO

CASO: Raíces distintas

Mostrar:

x(t) = k1 e^(m1 t) + k2 e^(m2 t)

Mostrar ecuación diferencial:

a x'' + b x' + c x = 0

--- GRAFICAR FUNCIÓN CARACTERÍSTICA ---

Crear valores xs en [-10 , 10]

Calcular f(xs)

Dibujar la función

Dibujar puntos (m1, 0) y (m2, 0)

Mostrar gráfica

--- GRAFICAR ITERACIONES M1 ---

Calcular f(x) para cada valor en its1

Dibujar curva de iteraciones con puntos

Mostrar gráfica

--- GRAFICAR ITERACIONES M2 ---

Igual procedimiento con its2

Mostrar gráfica

--- GRAFICAR SOLUCIÓN DE LA ECUACIÓN DIFERENCIAL ---

Crear vector t desde 0 a 10

SI raíz doble

x(t) = e^(m t) + t e^(m t)

SINO

x(t) = e^(m1 t) + e^(m2 t)

Dibujar la solución x(t)

Mostrar gráfica

SI ocurre cualquier error

Mostrar ventana de error

EULER.py

El programa implementa el Método de Euler para resolver numéricamente una ecuación diferencial ordinaria de la forma.

$$\frac{dx}{dt} = f(t, x)$$

Librerías utilizadas:

- NumPy: Cálculos matemáticos.
- Matplotlib: Gráficas.
- Tkinter: Interfaz gráfica.
- matplotlib.backends.backend_tkagg: Integración de Matplotlib con Tkinter.

Manual de código:

__init__(self, root):

- Configura la ventana principal.

```
self.root = root
self.root.title("Método de Euler - Solucionador EDO (t vs x)")
self.root.geometry("900x650")
```

- Crea los frames.

```
input_frame = ttk.LabelFrame(root, text="Parámetros de Entrada", padding="10")
input_frame.pack(side="top", fill="x", padx=10, pady=5)
```

- Inicializa widgets de entrada.

```
ttk.Label(input_frame, text="Función dx/dt = f(t,x):").grid(row=0, column=0, sticky="e")
ttk.Entry(input_frame, textvariable=self.func_str, width=30).grid(row=0, column=1, padx=5, pady=5)
```

- Crea tabla de iteraciones (Treeview).

```
table_frame = ttk.LabelFrame(results_frame, text="Tabla de Iteraciones")
table_frame.pack(side="left", fill="y", padx=5)
```

- Inicializa la gráfica de Matplotlib.

```
self.fig, self.ax = plt.subplots(figsize=(5, 4), dpi=100)
self.canvas = FigureCanvasTkAgg(self.fig, master=graph_frame)
self.canvas.get_tk_widget().pack(fill="both", expand=True)
```

evaluar_funcion(self, expr, t, x):

Evalúa la expresión ingresada (f(t,x)), en un entorno seguro, con funciones matemáticas permitidas. Se sustituyen t y x para evaluar la pendiente de Euler.

```
allowed_locals = {
    "sin": np.sin, "cos": np.cos, "tan": np.tan,
    "exp": np.exp, "sqrt": np.sqrt, "log": np.log,
    "pi": np.pi, "e": np.e
}
try:
    # CAMBIO CLAVE: El contexto ahora define 't' y 'x'
    return eval(expr, {"__builtins__": None}, {**allowed_locals, 't': t, 'x': x})
except Exception as e:
    raise ValueError(f"Error en la función: {e}")
```

calcular_euler(self):

Implementa el método de Euler:

$$x_{n+1} = x_n + h \cdot f(t_n, x_n)$$

- Leer valores de entrada.

```
f_str = self.func_str.get()
t0 = self.t0_val.get() # Variable independiente
x0 = self.x0_val.get() # Variable dependiente
h = self.h_val.get()
tf = self.tf_val.get() # Tiempo final
```

- Validar errores.

```
if h <= 0:
    messagebox.showerror("Error", "El paso h debe ser positivo.")
    return
```

- Ejecutar las iteraciones.

```
while t_actual < tf - 1e-9:
    try:
        # Pendiente depende ahora de t y x
        pendiente = self.evaluar_funcion(f_str, t_actual, x_actual)
    except ValueError as ve:
        messagebox.showerror("Error de Sintaxis", str(ve))
        return
```

```
x_siguiente = x_actual + h * pendiente
t_siguiente = t_actual + h
```

```
t_actual = t_siguiente
x_actual = x_siguiente
n += 1
```

- Llenar la tabla.

```
ts.append(t_actual)
xs.append(x_actual)
```

```
self.tree.insert("", "end", values=(n, f"{t_actual:.4f}", f"{x_actual:.6f}"))
```

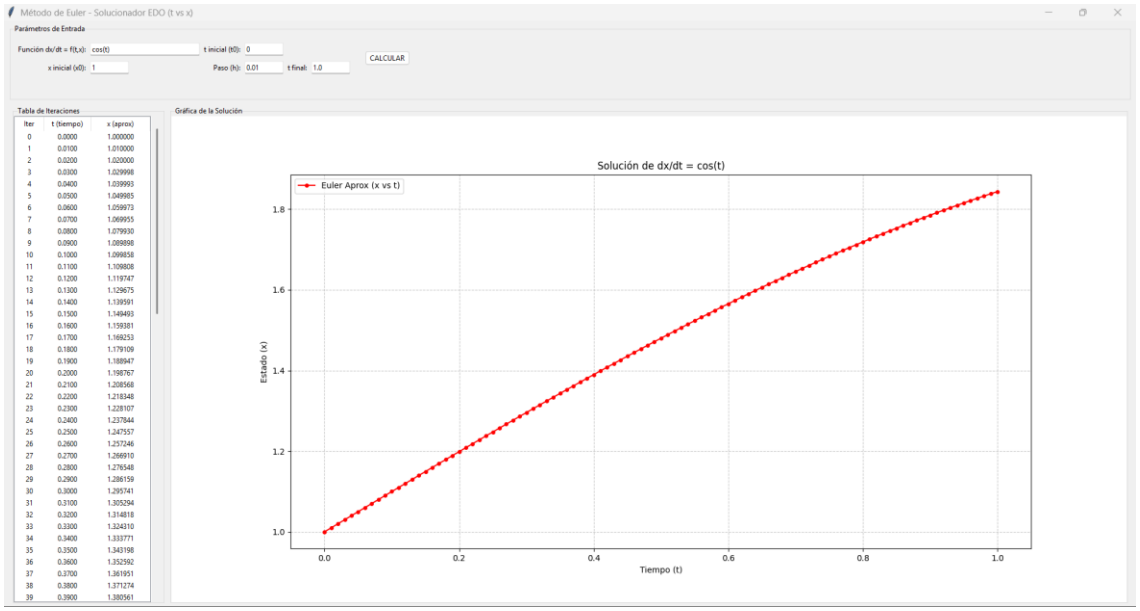
- Generar la gráfica.

```
self.ax.plot(ts, xs, 'r-o', label='Euler Aprox (x vs t)', markersize=4)
self.ax.set_title(f"Solución de dx/dt = {f_str}")
self.ax.set_xlabel("Tiempo (t)")
self.ax.set_ylabel("Estado (x)")
self.ax.grid(True, linestyle='--', alpha=0.7)
self.ax.legend()
```

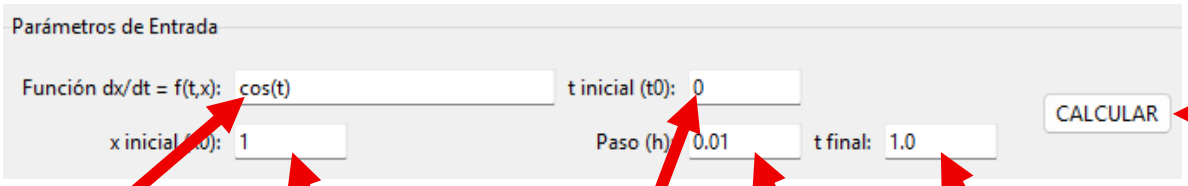
Manual de Usuario:

Este programa resuelve la ecuación diferencial dx/dt = f(t,x) usando el Método de Euler, mostrando:

- Una tabla con cada iteración
- Una gráfica de la solución aproximada



Parámetros de entrada:



Ingresar los datos, usar t y x como variables.

Ingresar condición inicial.

Ingresar tiempo donde comienza la integración.

Ingresar paso del método.

Hasta qué tiempo integrar.

- Evalúa la función
- Ejecuta Euler paso a paso
- Muestra la tabla de iteraciones
- Grafica x(t)

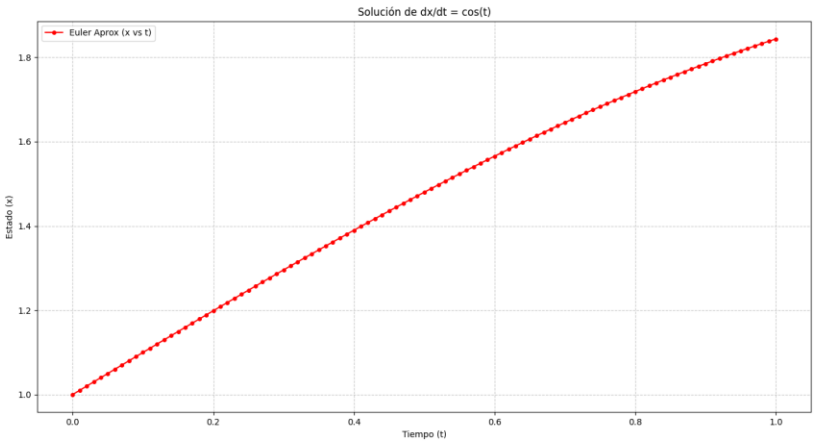
Tabla de Iteraciones

Iter	t (tiempo)	x (aprox)
62	0.6200	1.581961
63	0.6300	1.590100
64	0.6400	1.598180
65	0.6500	1.606201
66	0.6600	1.614162
67	0.6700	1.622062
68	0.6800	1.629900

Te genera 100 iteraciones, con una diferencia de 0.01 entra cada una. Donde te muestra el valor de 'x' aproximado, con su respectivo tiempo (t); hasta llegar a 1.

Y te genera una gráfica que representa la solución de la función:

- Línea roja con puntos (Euler)
- Eje horizontal: tiempo
- Eje vertical: solución aproximada



Pseudocódigo:

INICIAR ventana Tkinter

CREAR clase EulerApp con ventana principal

LLAMAR constructor:

- Configurar ventana
- Crear frame de entrada
- Crear campos: f(t,x), t0, x0, h, tf
- Crear botón CALCULAR
- Crear frame de resultados
- Crear tabla de iteraciones
- Crear gráfica embebida

MOSTRAR ventana

```
evaluar_funcion(expr, t, x):
RECIBIR expr, t, x
DEFINIR diccionario de funciones matemáticas permitidas
INTENTAR:
  EVALUAR expr reemplazando:
    t = valor de tiempo
    x = valor dependiente
REGRESAR resultado
```

```
calcular_euler():
LIMPIAR tabla y gráficas previas
LEER:
  f_str = función escrita
  t0 = tiempo inicial
  x0 = valor inicial
  h = paso
  tf = tiempo final
SI h <= 0:
  mostrar error
SI tf <= t0:
  fijar tf = t0 + 10*h
CREAR listas ts y xs con valores iniciales
INSERTAR en tabla la iteración 0 (t0, x0)
INICIALIZAR:
  t_actual = t0
  x_actual = x0
  n = 0
MIENTRAS t_actual < tf:
  CALCULAR pendiente = evaluar_funcion(f_str, t_actual, x_actual)
  t_siguiente = t_actual + h
  x_siguiente = x_actual + h * pendiente
  AGREGAR valores a ts y xs
  INSERTAR fila (n+1, t_siguiente, x_siguiente) en tabla
  ACTUALIZAR:
    t_actual = t_siguiente
    x_actual = x_siguiente
    n = n + 1
DIBUJAR gráfica:
  ts vs xs
  configurar títulos y ejes
ACTUALIZAR canvas de matplotlib
SI ocurre error:
  LANZAR "Error en la función"
```

eulermejorado.py

Este programa implementa una interfaz gráfica que resuelve ecuaciones diferenciales ordinarias (EDO) de primer orden de la forma usando el Método de Euler Mejorado (Heun).

$$\frac{dx}{dt} = f(t, x)$$

Manual de código:

__init__(self, root):

Configura:

- Ventana principal: Título (Método de Euler Mejorado (Heun) - [t vs x]), tamaño (950 × 650 px).

```
self.root = root
self.root.title("Método de Euler Mejorado (Heun) - [t vs x]")
self.root.geometry("950x650")
```

- Frame de Entrada:

  Incluye campos:

  - dx/dt = f(t,x): Ecuación diferencial a resolver.
  - t0: Tiempo inicial.
  - x0: Valor inicial de la variable dependiente.
  - h: Tamaño de paso.
  - t final: Tiempo final del intervalo.

```
self.func_str = tk.StringVar(value="t - x + 2") # Ejemplo: f(t,x)
self.t0_val = tk.DoubleVar(value=0.0) # Antes x0
self.x0_val = tk.DoubleVar(value=2.0) # Antes y0
self.h_val = tk.DoubleVar(value=0.1)
self.tf_val = tk.DoubleVar(value=1.0) # Antes xf
```

- Resultados:
  - Gráfica (matplotlib): Se presentan los puntos generados por Euler Mejorado con líneas y marcadores.

```
graph_frame = ttk.LabelFrame(results_frame, text="Gráfica")
graph_frame.pack(side="right", fill="both", expand=True, padx=5)

self.fig, self.ax = plt.subplots(figsize=(5, 4), dpi=100)
self.canvas = FigureCanvasTkAgg(self.fig, master=graph_frame)
self.canvas.get_tk_widget().pack(fill="both", expand=True)
```

- Tabla (ttk.Treeview):
  - Iteración n.
  - Tiempo t.
  - Aproximación x(t) con Euler Mejorado.

```
table_frame = ttk.LabelFrame(results_frame, text="Tabla de Iteraciones")
table_frame.pack(side="left", fill="y", padx=5)

columns = ("n", "t", "x_aprox")
self.tree = ttk.Treeview(table_frame, columns=columns, show="headings", height=20)
self.tree.heading("n", text="Iter")
self.tree.heading("t", text="t (tiempo)")
self.tree.heading("x_aprox", text="x (Mejorado)")

self.tree.column("n", width=40, anchor="center")
self.tree.column("t", width=80, anchor="center")
self.tree.column("x_aprox", width=120, anchor="center")

scrollbar = ttk.Scrollbar(table_frame, orient="vertical", command=self.tree.yview)
self.tree.configure(yscroll=scrollbar.set)
self.tree.pack(side="left", fill="both", expand=True)
scrollbar.pack(side="right", fill="y")
```

evaluar_funcion():

Evalúa la expresión ingresada por el usuario (f(t,x)).

- Se protege eval() removiendo builtins peligrosos.
- Se permiten funciones matemáticas: sin, cos, exp, sqrt, etc.
- Se inyectan valores para t y x.

```
allowed = {"sin": np.sin, "cos": np.cos, "tan": np.tan, "exp": np.exp, "sqrt": np.sqrt, "log": np.log,
           "pi": np.pi, "e": np.e}
# IMPORTANTE: Diccionario adaptado para recibir t y x
return eval(expr, {"__builtins__": None}, {**allowed, 't': t, 'x': x})
```

calcular_heun():

Este método ejecuta el método de Euler Mejorado:

1. Obtener valores iniciales.

```
f_str = self.func_str.get()
t0 = self.t0_val.get()
x0 = self.x0_val.get()
h = self.h_val.get()
tf = self.tf_val.get()
```

2. Insertar valores iniciales en la tabla.

```
ts = [t0]
xs = [x0]
t = t0
x = x0
n = 0

self.tree.insert("", "end", values=(n, f"{t:.4f}", f"{x:.6f}"))
```

3. Mientras $t < tf$:

```
while t < tf - 1e-9:
```

- $k_1 = f(t, x)$

```
k1 = self.evaluar_funcion(f_str, t, x)
```

- Predecir x con Euler simple: $x = x + hk_1$

```
x_star = x + h * k1
```

- Promediar pendientes: $\frac{k_1 + k_2}{2}$

```
pendiente_promedio = (k1 + k2) / 2
```

- Obtener nuevo valor: $x_{n+1} = x + h(\frac{k_1 + k_2}{2})$

```
x_next = x + h * pendiente_promedio
```

4. Mostrar gráfica.

```
self.ax.plot(ts, xs, 'g-o', label='Euler Mejorado (Heun)', markersize=4)
self.ax.set_title(f"Solución: {f_str}")
self.ax.set_xlabel("Tiempo (t)")
self.ax.set_ylabel("Estado (x)")
self.ax.grid(True, linestyle='--', alpha=0.7)
self.ax.legend()
self.canvas.draw()
```

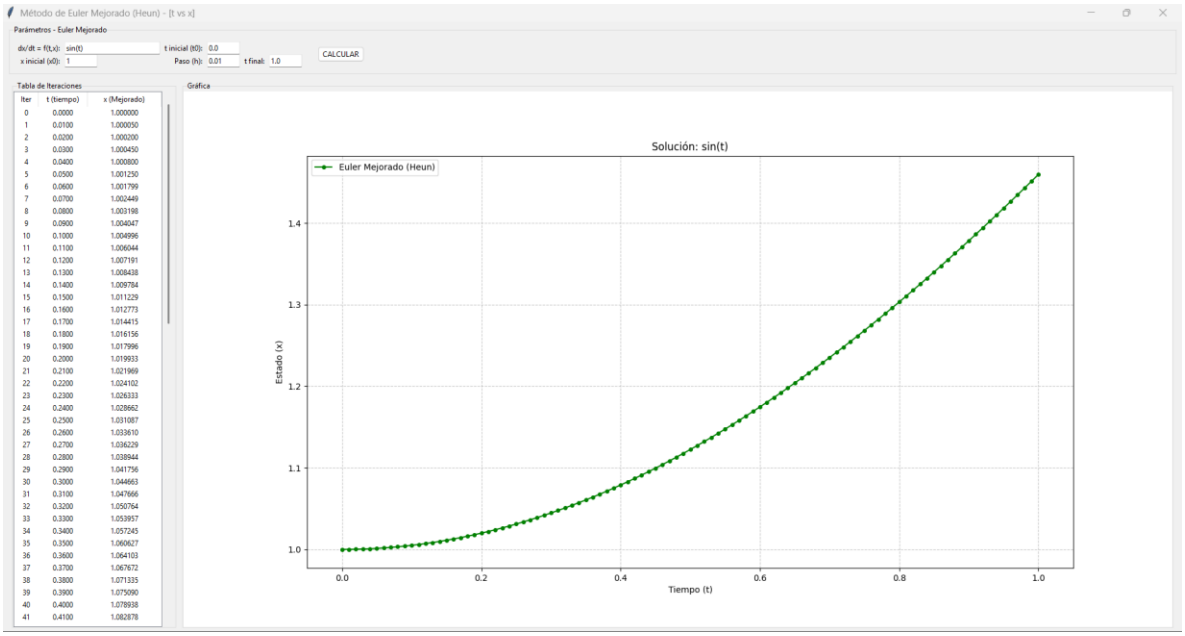
Maneja errores con try/except.

```
except Exception as e:
    messagebox.showerror("Error", f"Error de cálculo: {e}")
```

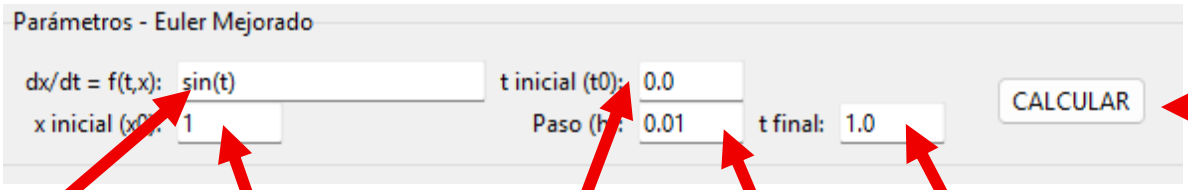
Manual de Usuario:

Este software permite resolver ecuaciones diferenciales de primer orden mediante el Método de Euler Mejorado (Heun) y visualizar:

- Tabla de cálculos iterativos
- Gráfica de la curva aproximada



Parámetros de entrada:



Ingresar los datos, usar t y x como variables.

Ingresar condición inicial.

Ingresar tiempo donde comienza la integración.

Ingresar paso del método.

Hasta qué tiempo integrar.

- Genera iteraciones.
- Llena la tabla.
- Muestra la gráfica (x vs t)

Tabla de Iteraciones

Iter	t (tiempo)	x (Mejorado)
59	0.5900	1.169058
60	0.6000	1.174663
61	0.6100	1.180350
62	0.6200	1.186120
63	0.6300	1.191971
64	0.6400	1.197903
65	0.6500	1.203915

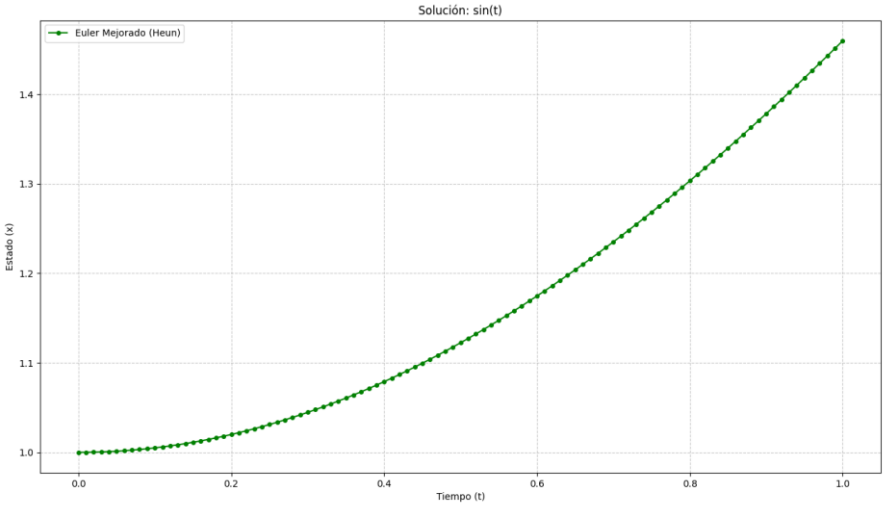
Te genera 100 iteraciones, con una diferencia de 0.01 entra cada una. Donde te muestra el valor de 'x' mejorado (Usando la fórmula de Euler mejorada.), con su respectivo tiempo (t); hasta llegar a 1.

Muestra por fila:

- Número de iteración n.
- Tiempo t.
- Valor x(t) usando Heun.

Gráfica

- Línea verde: solución aproximada.
- Puntos marcados: iteraciones del método.



Pseudocódigo:

INICIAR GUI

CREAR entrada función f(t,x)

CREAR entrada t0

CREAR entrada x0

CREAR entrada h

CREAR entrada tf

CREAR botón CALCULAR -> ejecutar calcular_heun()

CREAR tabla de iteraciones

CREAR panel para gráfica

evaluar_funcion(expr, t, x):

allowed ← diccionario con funciones matemáticas

evaluar expr usando eval con t y x

retornar resultado

calcular_heun():

LIMPIAR tabla

LIMPIAR gráfica

LEER f_str, t0, x0, h, tf

SI h ≤ 0 → error

SI tf ≤ t0 → error

t ← t0

x ← x0

n ← 0

INSERTAR fila inicial (n, t, x)

CREAR listas ts = [t], xs = [x]

MIENTRAS t < tf:

  k1 ← evaluar_funcion(f_str, t, x)

  x_star ← x + h * k1

  t_next ← t + h

  k2 ← evaluar_funcion(f_str, t_next, x_star)

  pendiente_promedio ← (k1 + k2) / 2

  x_next ← x + h * pendiente_promedio

  t ← t_next

  x ← x_next

  n ← n + 1

  AGREGAR t y x a listas ts y xs

  INSERTAR fila a tabla

FIN MIENTRAS

GRAFICAR ts vs xs

MOSTRAR gráfica

RK4.py

Este programa resuelve numéricamente ecuaciones diferenciales de primer orden del tipo:

$$\frac{dx}{dt} = f(t, x)$$

usando el método Runge-Kutta clásico de 4to orden (RK4).

Manual de código:

__init__(self, root):

Configura:

- Ventana principal: Título (Método Runge-Kutta 4 (RK4) - [t vs x]), tamaño (950 × 650 px).

```
self.root = root
self.root.title("Método Runge-Kutta 4 (RK4) - [t vs x]")
self.root.geometry("950x650")
```

- Frame de Entrada:

  Incluye campos:

  - dx/dt = f(t,x): Ecuación diferencial a resolver.
  - t0: Tiempo inicial.
  - x0: Valor inicial de la variable dependiente.
  - h: Tamaño de paso.
  - t final: Tiempo final del intervalo.

```
self.func_str = tk.StringVar(value="t - x + 2") # Ejemplo: f(t,x)
self.t0_val = tk.DoubleVar(value=0.0) # Antes x0
self.x0_val = tk.DoubleVar(value=2.0) # Antes y0
self.h_val = tk.DoubleVar(value=0.1)
self.tf_val = tk.DoubleVar(value=1.0) # Antes xf
```

- Resultados:
  - Gráfica (matplotlib): Se presentan los puntos generados por Euler Mejorado con líneas y marcadores.

```
graph_frame = ttk.LabelFrame(results_frame, text="Gráfica")
graph_frame.pack(side="right", fill="both", expand=True, padx=5)

self.fig, self.ax = plt.subplots(figsize=(5, 4), dpi=100)
self.canvas = FigureCanvasTkAgg(self.fig, master=graph_frame)
self.canvas.get_tk_widget().pack(fill="both", expand=True)
```

- Tabla (ttk.Treeview):
  - Iteración n.
  - Tiempo t.
  - x_rk4 (aproximación de x).

```
table_frame = ttk.LabelFrame(results_frame, text="Tabla de Resultados")
table_frame.pack(side="left", fill="y", padx=5)

columns = ("n", "t", "x_rk4")
self.tree = ttk.Treeview(table_frame, columns=columns, show="headings", height=20)
self.tree.heading("n", text="Iter")
self.tree.heading("t", text="t (tiempo)")
self.tree.heading("x_rk4", text="x (RK4)")
```

evaluar_funcion():

Evalúa la expresión ingresada por el usuario (f(t,x)).

- Se protege eval() removiendo builtins peligrosos.
- Se permiten funciones matemáticas: sin, cos, exp, sqrt, etc.
- Se inyectan valores para t y x.

```
allowed = {"sin": np.sin, "cos": np.cos, "tan": np.tan, "exp": np.exp, "sqrt": np.sqrt, "log": np.log,
           "pi": np.pi, "e": np.e}
# Evalúa f(t,x) de forma segura
return eval(expr, {"__builtins__": None}, {**allowed, 't': t, 'x': x})
```

calcular_rk4(self):

Este método realiza:

1. Limpieza de tabla y gráfica.

```
for item in self.tree.get_children():
    self.tree.delete(item)
self.ax.clear()
```

2. Lectura de parámetros.

```
f_str = self.func_str.get()
t0 = self.t0_val.get()
x0 = self.x0_val.get()
h = self.h_val.get()
tf = self.tf_val.get()
```

3. Validación:
    - h > 0
    - tf > t0

```
while t < tf - 1e-9:
```

4. Registro de condición inicial.

```
ts = [t0]
xs = [x0]
t = t0
x = x0
n = 0
```

5. Ejecución del método RK4.

```
k1 = self.evaluar_funcion(f_str, t, x)
k2 = self.evaluar_funcion(f_str, t + 0.5 * h, x + 0.5 * h * k1)
k3 = self.evaluar_funcion(f_str, t + 0.5 * h, x + 0.5 * h * k2)
k4 = self.evaluar_funcion(f_str, t + h, x + h * k3)
x_next = x + (h / 6.0) * (k1 + 2 * k2 + 2 * k3 + k4)
t_next = t + h
```

6. Llenado de tabla.

```
ts.append(t)
xs.append(x)
self.tree.insert("", "end", values=(n, f"{t:.4f}", f"{x:.6f}"))
```

7. Gráfica de los resultados.

```
self.ax.plot(ts, xs, 'r-o', label='Método RK4', markersize=4)
self.ax.set_title(f"Solución RK4: {f_str}")
self.ax.set_xlabel("Tiempo (t)")
self.ax.set_ylabel("Estado (x)")
self.ax.grid(True, linestyle='--', alpha=0.7)
self.ax.legend()
self.canvas.draw()
```

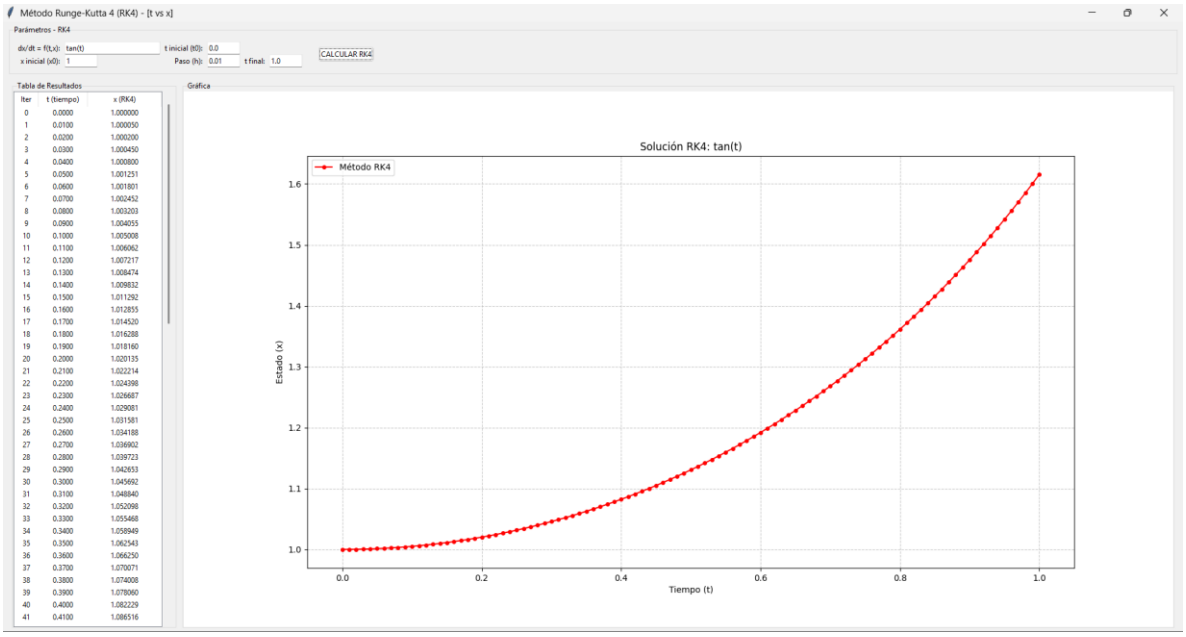
Núcleo del método RK4:

Se calculan los cuatro coeficientes:

$$
\begin{aligned}
k_1 &= f(t,\ x) \\
k_2 &= f\left(t + \frac{h}{2},\ x + \frac{h}{2}k_1\right) \\
k_3 &= f\left(t + \frac{h}{2},\ x + \frac{h}{2}k_2\right) \\
k_4 &= f(t + h,\ x + hk_3)
\end{aligned}
$$

$$
x_{n+1} = x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)
$$

Manual de Usuario:



Parámetros de entrada:

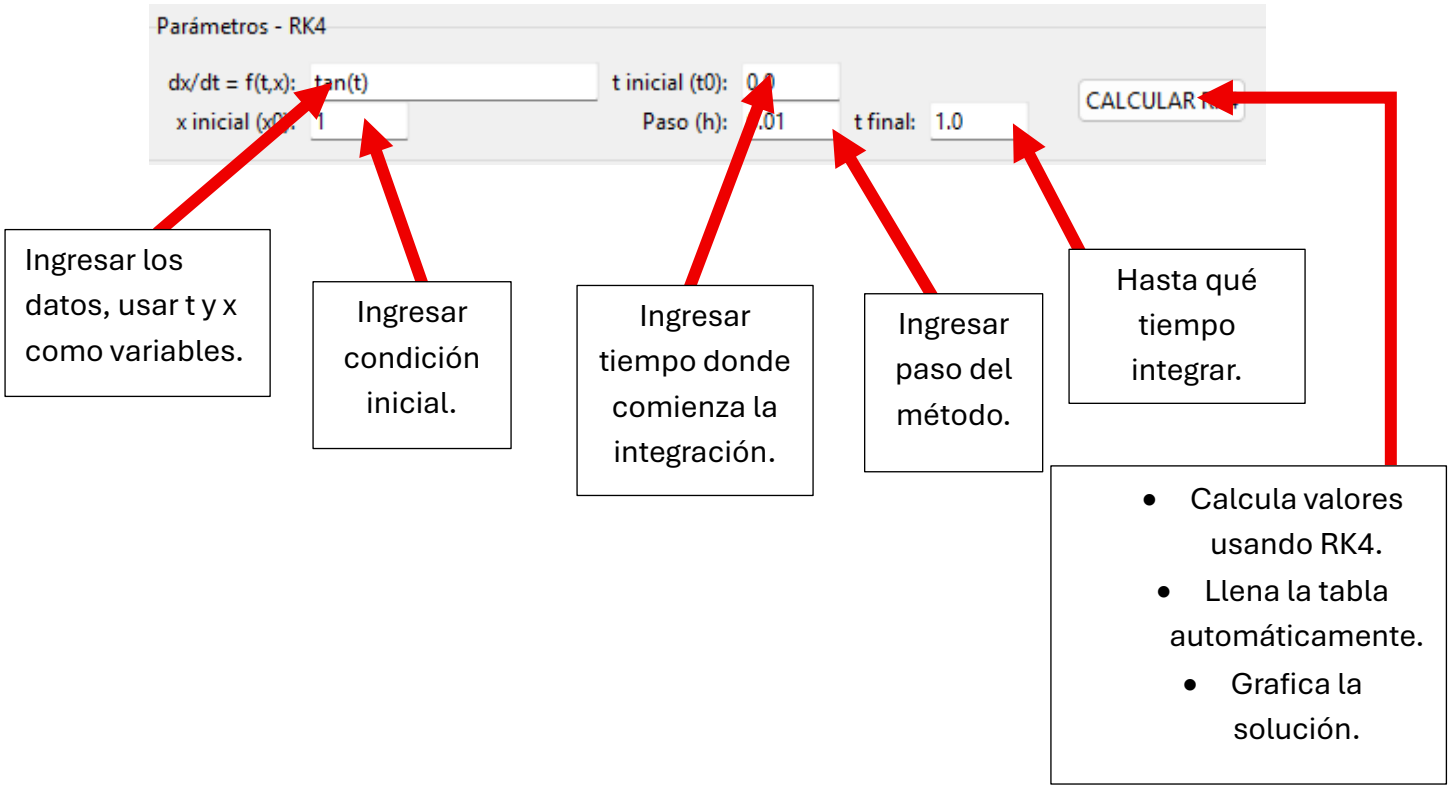


Tabla de Resultados

Iter	t (tiempo)	x (RK4)
59	0.5900	1.185197
60	0.6000	1.191965
61	0.6100	1.198880
62	0.6200	1.205944
63	0.6300	1.213159
64	0.6400	1.220527
65	0.6500	1.228051
66	0.6600	1.235732

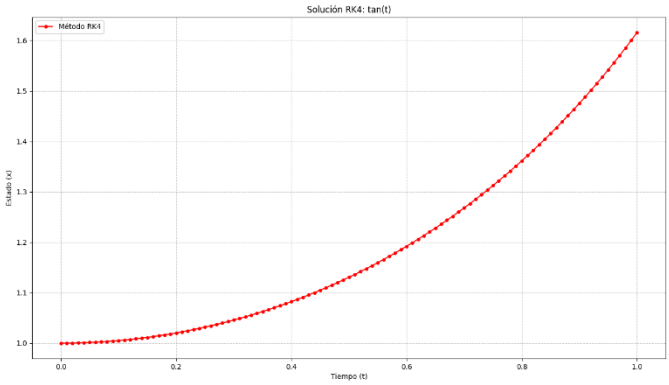
Te genera 100 iteraciones, con una diferencia de 0.01 entra cada una. Donde te muestra el valor de 'x' (Usando la fórmula de RK4.), con su respectivo tiempo (t); hasta llegar a 1.

Muestra por fila:

- Número de iteración n.
- Tiempo t.
- Valor x(t) usando RK4.

Gráfica

- Línea roja = aproximación RK4
- Se muestra la trayectoria x vs t



Pseudocódigo:

INICIAR ventana principal

CREAR campos para: f(t,x), t0, x0, h, tf

CREAR botón CALCULAR RK4 --> calcular_rk4()

CREAR tabla para mostrar iteraciones

CREAR área para gráfica

evaluar_funcion():

allowed ← diccionario con funciones matemáticas

evaluar expr con eval sin builtins

usar t y x actuales

retornar resultado

calcular_rk4 ():

LIMPIAR tabla

LIMPIAR gráfica

LEER f_str, t0, x0, h, tf

SI h <= 0 → error

SI tf <= t0 → error

t ← t0

x ← x0

n ← 0

INSERTAR fila (n, t, x)

ts ← lista con t0

xs ← lista con x0

MIENTRAS t < tf:

  k1 ← f(t, x)

  k2 ← f(t + h/2, x + h/2 * k1)

  k3 ← f(t + h/2, x + h/2 * k2)

  k4 ← f(t + h, x + h * k3)

  x_next ← x + (h / 6) * (k1 + 2k2 + 2k3 + k4)

  t_next ← t + h

  t ← t_next

  x ← x_next

  n ← n + 1

  AGREGAR t y x a ts y xs

  INSERTAR fila en tabla

FIN MIENTRAS

GRAFICAR ts vs xs

ACTUALIZAR canvas

REFERENCIAS:

Khan Academy. (s.f.). *Newton's method*. Recuperado de https://www.khanacademy.org/math/calculus-1/cs1-applications-newton

GeeksforGeeks. (2021). *Newton Raphson Method*. Recuperado de https://www.geeksforgeeks.org/newton-raphson-method/

Khan Academy. (s.f.). *Trapezoidal rule*. Recuperado de https://www.khanacademy.org/math/ap-calculus-ab/ab-integration-new/ab-6-11

Brilliant.org. (s.f.). *Newton's Method*. Recuperado de https://brilliant.org/wiki/newtons-method/

Khan Academy. (s.f.). *Euler's method*. Recuperado de https://www.khanacademy.org/math/differential-equations/first-order-differential-equations/eulers-method

LibreTexts. (2023). *Euler's Method*. Recuperado de https://math.libretexts.org/Bookshelves/Calculus/Book%3A_Calculus_(OpenStax)/09%3A_Differential_Equations/9.2%3A_Euler's_Method

GeeksforGeeks. (2020). *Runge–Kutta method*. Recuperado de https://www.geeksforgeeks.org/runge-kutta-method/

Programación ATS. (2020). *Método de Newton-Raphson en Python* [Video]. YouTube. https://www.youtube.com/watch?v=mhBe7m5lB8s

Computing for Engineers. (2022). Trapezoidal Rule in Python [Video]. YouTube.

https://www.youtube.com/watch?v=Gk2v_SXWnYQ

Data Science Damián. (2021). *Cómo encontrar raíces en Python (Newton)* [Video]. YouTube.
https://www.youtube.com/watch?v=uqbeqv1jH78

Engineering with Python. (2020). *Euler's Method in Python* [Video]. YouTube. https://www.youtube.com/watch?v=xsM8Pee4JyI

Brian Murphy. (2021). *Runge-Kutta 4 (RK4) in Python Step-by-Step* [Video]. YouTube. https://www.youtube.com/watch?v=9oK_UA6xVWM