```
import tkinter as tk
from tkinter import ttk
from tkinter import messagebox
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg,
NavigationToolbar2Tk
import re
import csv
import time
import os
import sys  # Import sys for path handling
from PIL import Image, ImageTk


class ODESolverApp:
    def __init__(self, master):
        self.master = master
        self.master.title("NUMERICAL METHODS FOR SOLVING ODES")

        self.master.withdraw()

        self._show_splash_screen()

        self.ode_function_str = tk.StringVar(master, value="")
        self.x0_str = tk.StringVar(master, value="")
        self.y0_str = tk.StringVar(master, value="")
        self.xf_str = tk.StringVar(master, value="")
        self.h_str = tk.StringVar(master, value="")
        self.selected_method = tk.StringVar(master, value="Euler")
        self.selected_method.trace_add("write", self._on_method_change)

        self.x_values = []
        self.y_values = []

        self.csv_file_path = "ode_solution_data.csv"
        self.record_counter = 0

        self.entry_widgets = []

        self.master.after(2000, self._create_widgets)

    def _show_splash_screen(self):
        self.splash_screen = tk.Toplevel(self.master)
        self.splash_screen.overrideredirect(True)

        screen_width = self.master.winfo_screenwidth()
        screen_height = self.master.winfo_screenheight()
        splash_width = 1200
        splash_height = 750
        x = (screen_width // 2) - (splash_width // 2)
        y = (screen_height // 2) - (splash_height // 2)

self.splash_screen.geometry(f"{splash_width}x{splash_height}+{x}+{y}")
```

```
        splash_label = ttk.Label(self.splash_screen,
                                 text="NUMERICAL METHODS FOR SOLVING
ODES\n\nLoading...",
                                 font=("Helvetica", 16, "bold"),
                                 background="#3498db",
                                 foreground="white",
                                 anchor="center",
                                 justify="center")
        splash_label.pack(expand=True, fill="both")

        try:
            # Determine the base path for the application
            # This helps when running as a compiled executable (e.g.,
with PyInstaller)
            if getattr(sys, 'frozen', False):
                # If the application is frozen (e.g., PyInstaller)
                base_path = sys._MEIPASS
            else:
                # If running as a normal Python script
                base_path = os.path.dirname(os.path.abspath(__file__))

            image_path = os.path.join(base_path, "icon.jpg")

            original_image = Image.open(image_path)
            resized_image = original_image.resize((500, 500),
Image.Resampling.LANCZOS)  # Adjusted to 500x100
            self.splash_image = ImageTk.PhotoImage(resized_image)
            image_label = tk.Label(self.splash_screen,
image=self.splash_image, background="#3498db")
            image_label.pack(pady=10)
            splash_label.pack_forget()
            image_label.pack(side="top", pady=20)
            splash_label.pack(side="bottom", expand=True)
            print(f"Successfully loaded image from: {image_path}")  #
Debug print
        except ImportError:
            print(
                "Pillow not installed. Cannot display image on splash
screen. Please install it using 'pip install Pillow'.")
        except FileNotFoundError:
            print(f"Error: icon.jpg not found at {image_path}. Ensure
it's in the correct directory.")
        except Exception as e:
            print(f"An unexpected error occurred during image loading:
{e}")

        self.splash_screen.after(2000, self._hide_splash_screen)

    def _hide_splash_screen(self):
        self.splash_screen.destroy()
        self.master.deiconify()
        self.master.focus_set()
```

```
    def _create_widgets(self):
        self.master.grid_rowconfigure(0, weight=0)
        self.master.grid_rowconfigure(1, weight=0)
        self.master.grid_rowconfigure(2, weight=0)
        self.master.grid_rowconfigure(3, weight=1)
        self.master.grid_rowconfigure(4, weight=0)

        self.master.grid_columnconfigure(0, weight=1)
        self.master.grid_columnconfigure(1, weight=1)

        # Main Input Frame
        input_frame = ttk.LabelFrame(self.master, text="ODE Parameters")
        input_frame.grid(row=0, column=0, columnspan=2, padx=10, pady=5,
sticky="ew")

        # Configure columns within input_frame for better horizontal
distribution
        input_frame.grid_columnconfigure(1, weight=1)
        input_frame.grid_columnconfigure(3, weight=1)
        input_frame.grid_columnconfigure(4, weight=0)  # Tips frame
column should not expand horizontally

        # Row 0: ODE function and Input Tips
        ttk.Label(input_frame, text="dy/dx = f(x, y):").grid(row=0,
column=0, padx=5, pady=2, sticky="w")
        ode_entry = ttk.Entry(input_frame,
textvariable=self.ode_function_str, width=30)
        ode_entry.grid(row=0, column=1, padx=5, pady=2, sticky="ew")
        self.entry_widgets.append(ode_entry)

        # Input Tips Frame
        tips_frame = ttk.LabelFrame(input_frame, text="Input Tips",
style="Tips.TLabelframe")
        tips_frame.grid(row=0, column=4, rowspan=3, padx=5, pady=2,
                        sticky="nsew")  # Place tips in column 4,
spanning 3 rows

        # Style for the new tips frame
        style = ttk.Style()
        style.configure("Tips.TLabelframe", background="#e0f2f7")  #
Light blue background
        style.configure("Tips.TLabelframe.Label", background="#e0f2f7")
# Ensure label background matches

        ttk.Label(tips_frame, text="â€¢ Power: Use `^` (e.g., `x^2`).\n"
                                  "â€¢ Implicit Multiplication:
Supported (e.g.,`xy`).\n"
                                  "â€¢ Exponential: Use `exp(x)` for
e^x.\n"
                                  "â€¢ Logarithm: Use `log(x)` for
natural log, `log10(x)` for base-10 log.\n"
                                  "â€¢ Trigonometric: Use `sin(x)`,
`cos(x)`, `tan(x)`.\n"
```

```
                                "â€¢ Other Functions: `sqrt()`,
`abs()`, `pi`.",
                  justify=tk.LEFT,
                  background="#e0f2f7"  # Set background for the label
inside the tips frame
                  ).pack(padx=5, pady=5, anchor="nw")

        # Row 1: Initial X and Final X
        ttk.Label(input_frame, text="Initial x (x0):").grid(row=1,
column=0, padx=5, pady=2, sticky="w")
        x0_entry = ttk.Entry(input_frame, textvariable=self.x0_str,
width=15)
        x0_entry.grid(row=1, column=1, padx=5, pady=2, sticky="w")
        self.entry_widgets.append(x0_entry)

        ttk.Label(input_frame, text="Final x (xf):").grid(row=1,
column=2, padx=5, pady=2, sticky="w")
        xf_entry = ttk.Entry(input_frame, textvariable=self.xf_str,
width=15)
        xf_entry.grid(row=1, column=3, padx=5, pady=2, sticky="w")
        self.entry_widgets.append(xf_entry)

        # Row 2: Initial Y and Step Size H
        ttk.Label(input_frame, text="Initial y (y0):").grid(row=2,
column=0, padx=5, pady=2, sticky="w")
        y0_entry = ttk.Entry(input_frame, textvariable=self.y0_str,
width=15)
        y0_entry.grid(row=2, column=1, padx=5, pady=2, sticky="w")
        self.entry_widgets.append(y0_entry)

        ttk.Label(input_frame, text="Step Size (h):").grid(row=2,
column=2, padx=5, pady=2, sticky="w")
        h_entry = ttk.Entry(input_frame, textvariable=self.h_str,
width=15)
        h_entry.grid(row=2, column=3, padx=5, pady=2, sticky="w")
        self.entry_widgets.append(h_entry)

        # Bind <Return> event to all entry widgets
        for i, entry in enumerate(self.entry_widgets):
            if i < len(self.entry_widgets) - 1:
                entry.bind("<Return>", lambda event,
next_entry=self.entry_widgets[i + 1]: next_entry.focus_set())
            else:
                entry.bind("<Return>", lambda event: self.solve_ode())

        # Method Selection Frame
        method_frame = ttk.LabelFrame(self.master, text="Select Method")
        method_frame.grid(row=1, column=0, columnspan=2, padx=10, pady=5,
sticky="ew")

        method_frame.grid_columnconfigure(0, weight=1)
        method_frame.grid_columnconfigure(1, weight=1)
        method_frame.grid_columnconfigure(2, weight=1)
        method_frame.grid_columnconfigure(3, weight=1)
```

```
        ttk.Radiobutton(method_frame, text="Euler's Method",
variable=self.selected_method, value="Euler").grid(row=0,

column=0,

sticky="w",

padx=5,

pady=2)
        ttk.Radiobutton(method_frame, text="Improved Euler (Heun's
Method)", variable=self.selected_method,
                        value="Improved Euler").grid(row=0, column=1,
sticky="w", padx=5, pady=2)
        ttk.Radiobutton(method_frame, text="Runge-Kutta (RK4)",
variable=self.selected_method, value="RK4").grid(row=0,

column=2,

sticky="w",

padx=5,

pady=2)
        ttk.Radiobutton(method_frame, text="All Methods (Compare)",
variable=self.selected_method, value="All").grid(
            row=0, column=3, sticky="w", padx=5, pady=2)

        # Control Buttons Frame
        button_frame = ttk.Frame(self.master)
        button_frame.grid(row=2, column=0, columnspan=2, padx=10, pady=5)

        button_frame.grid_columnconfigure(0, weight=1)
        button_frame.grid_columnconfigure(1, weight=1)
        button_frame.grid_columnconfigure(2, weight=1)
        ttk.Button(button_frame, text="Solve",
command=self.solve_ode).grid(row=0, column=0, padx=5, pady=2)
        ttk.Button(button_frame, text="Clear",
command=self.clear_results).grid(row=0, column=1, padx=5, pady=2)
        ttk.Button(button_frame, text="Exit",
command=self.master.quit).grid(row=0, column=2, padx=5, pady=2)

        # Plotting Area
        self.fig, self.ax = plt.subplots(figsize=(6, 4))
        self.canvas = FigureCanvasTkAgg(self.fig, master=self.master)
        self.canvas_widget = self.canvas.get_tk_widget()
        self.canvas_widget.grid(row=3, column=0, padx=10, pady=5,
sticky="nsew")

        self.results_tree = ttk.Treeview(self.master)
        self._configure_results_tree(self.selected_method.get())
        self.results_tree.grid(row=3, column=1, padx=10, pady=5,
sticky="nsew")
```

```
        toolbar_frame = ttk.Frame(self.master)
        toolbar_frame.grid(row=4, column=0, columnspan=2, sticky="ew",
padx=10, pady=(0, 5))

        self.toolbar = NavigationToolbar2Tk(self.canvas, toolbar_frame)
        self.toolbar.update()
        self.toolbar.pack(side="top", fill="both", expand=True)

    def _on_method_change(self, *args):
        self._configure_results_tree(self.selected_method.get())
        self._clear_plot_and_table()

    def _configure_results_tree(self, method_type):
        self.results_tree.delete(*self.results_tree.get_children())
        self.results_tree["columns"] = ()
        self.results_tree.heading("#0", text="")
        self.results_tree.column("#0", width=0, stretch=tk.NO)

        style = ttk.Style()
        style.configure("Custom.Treeview.Heading", background="darkblue",
foreground="white",
                        font=('TkDefaultFont', 11, 'bold'))

        if method_type == "All":
            self.results_tree["columns"] = ("x", "euler_y",
"improved_euler_y", "rk4_y")
            self.results_tree.heading("x", text="X Value")
            self.results_tree.heading("euler_y", text="Euler Y")
            self.results_tree.heading("improved_euler_y", text="Improved
Euler Y")
            self.results_tree.heading("rk4_y", text="RK4 Y")
            self.results_tree.column("x", width=100, anchor="center")
            self.results_tree.column("euler_y", width=100,
anchor="center")
            self.results_tree.column("improved_euler_y", width=100,
anchor="center")
            self.results_tree.column("rk4_y", width=100, anchor="center")
        else:
            self.results_tree["columns"] = ("x", "y")
            self.results_tree.heading("x", text="X Value")
            self.results_tree.heading("y", text="Y Value")
            self.results_tree.column("x", width=100, anchor="center")
            self.results_tree.column("y", width=100, anchor="center")

        self.results_tree.config(show="headings")

    def preprocess_ode_string(self, ode_str):
        processed_str = ode_str.replace('^', '**')

        processed_str = re.sub(r'(\d(?:\.\d*)?)([xy])', r'\1*\2',
processed_str)
```

```
        processed_str = re.sub(r'([xy])(\d(?:\.\d*)?)', r'\1*\2',
processed_str)

        processed_str = re.sub(r'([x])([y])', r'\1*\2', processed_str)
        processed_str = re.sub(r'([y])([x])', r'\1*\2', processed_str)

        processed_str = re.sub(r'(\))([xy\d(])', r'\1*\2', processed_str)

        processed_str = re.sub(r'([xy\d])(\()', r'\1*\2', processed_str)

        return processed_str

    def _parse_input_function(self, x, y):
        raw_ode_string = self.ode_function_str.get()

        processed_ode_string = self.preprocess_ode_string(raw_ode_string)

        try:
            allowed_globals = {
                "__builtins__": None,
                "x": x,
                "y": y,
                "np": np,
                "e": np.e,
                "pi": np.pi,
                "exp": np.exp,
                "log": np.log,
                "log10": np.log10,
                "sin": np.sin,
                "cos": np.cos,
                "tan": np.tan,
                "arcsin": np.arcsin,
                "arccos": np.arccos,
                "arctan": np.arctan,
                "sinh": np.sinh,
                "cosh": np.cosh,
                "tanh": np.tanh,
                "sqrt": np.sqrt,
                "abs": np.abs,
                "ceil": np.ceil,
                "floor": np.floor,
                "round": np.round,
            }
            return eval(processed_ode_string, allowed_globals)
        except SyntaxError:
            messagebox.showerror("Input Error",
                                 "Syntax Error in ODE function. Please
check your expression for correct mathematical syntax.")
            return None
        except NameError as ne:
            messagebox.showerror("Input Error",
                                 f"Undefined variable or function in ODE:
{ne}. Ensure you are only using 'x', 'y', and supported functions (e.g.,
'exp(x)', 'sin(y)').")
```

```
            return None
        except Exception as e:
            messagebox.showerror("Error", f"An unexpected error occurred
during ODE evaluation: {e}")
            return None

    def _euler_method(self, x0, y0, xf, h):
        x_vals = [x0]
        y_vals = [y0]

        current_x = x0
        current_y = y0

        while current_x < xf - 1e-9:
            f_val = self._parse_input_function(current_x, current_y)
            if f_val is None: return [], []

            step_to_take = h
            if current_x + h > xf + 1e-9:
                step_to_take = xf - current_x

            if step_to_take <= 0:
                break

            next_y = current_y + step_to_take * f_val
            next_x = current_x + step_to_take

            x_vals.append(next_x)
            y_vals.append(next_y)

            current_x = next_x
            current_y = next_y

            if abs(current_x - xf) < 1e-9:
                break

        if not x_vals and x0 == xf:
            x_vals = [x0]
            y_vals = [y0]

        return x_vals, y_vals

    def _improved_euler_method(self, x0, y0, xf, h):
        x_vals = [x0]
        y_vals = [y0]

        current_x = x0
        current_y = y0

        while current_x < xf - 1e-9:
            f_val = self._parse_input_function(current_x, current_y)
            if f_val is None: return [], []

            step_to_take = h
```

```
            if current_x + h > xf + 1e-9:
                step_to_take = xf - current_x

            if step_to_take <= 0:
                break

            y_predictor = current_y + step_to_take * f_val
            f_next_val = self._parse_input_function(current_x +
step_to_take, y_predictor)
            if f_next_val is None: return [], []

            next_y = current_y + (step_to_take / 2) * (f_val +
f_next_val)
            next_x = current_x + step_to_take

            x_vals.append(next_x)
            y_vals.append(next_y)

            current_x = next_x
            current_y = next_y

            if abs(current_x - xf) < 1e-9:
                break

        if not x_vals and x0 == xf:
            x_vals = [x0]
            y_vals = [y0]

        return x_vals, y_vals

    def _runge_kutta_4th_order(self, x0, y0, xf, h):
        x_vals = [x0]
        y_vals = [y0]

        current_x = x0
        current_y = y0

        while current_x < xf - 1e-9:
            k1 = self._parse_input_function(current_x, current_y)
            if k1 is None: return [], []

            step_to_take = h
            if current_x + h > xf + 1e-9:
                step_to_take = xf - current_x

            if step_to_take <= 0:
                break

            k2 = self._parse_input_function(current_x + step_to_take / 2,
current_y + (step_to_take / 2) * k1)
            if k2 is None: return [], []
            k3 = self._parse_input_function(current_x + step_to_take / 2,
current_y + (step_to_take / 2) * k2)
            if k3 is None: return [], []
```

```
            k4 = self._parse_input_function(current_x + step_to_take,
current_y + step_to_take * k3)
            if k4 is None: return [], []

            next_y = current_y + (h / 6) * (k1 + 2 * k2 + 2 * k3 + k4)
            next_x = current_x + step_to_take

            x_vals.append(next_x)
            y_vals.append(next_y)

            current_x = next_x
            current_y = next_y

            if abs(current_x - xf) < 1e-9:
                break

        if not x_vals and x0 == xf:
            x_vals = [x0]
            y_vals = [y0]

        return x_vals, y_vals

    def solve_ode(self):
        if not self.ode_function_str.get():
            messagebox.showerror("Input Error", "Please enter the ODE
function (dy/dx = f(x, y)).")
            return

        try:
            x0 = float(self.x0_str.get())
            y0 = float(self.y0_str.get())
            xf = float(self.xf_str.get())
            h = float(self.h_str.get())

            if h <= 0 or xf < x0:
                messagebox.showerror("Input Error",
                                     "Step size must be positive and
final x must be greater than or equal to initial x.")
                return

            method_selection = self.selected_method.get()

            all_methods_plot_data = []
            all_methods_table_data_combined = []

            self._clear_plot_and_table()

            self.record_counter += 1

            if method_selection == "Euler":
                start_time = time.time()
                euler_x, euler_y = self._euler_method(x0, y0, xf, h)
                end_time = time.time()
                if not euler_x: return
```

```
                all_methods_plot_data.append(("Euler", euler_x, euler_y))
                all_methods_table_data_combined = [("Euler", euler_x,
euler_y)]

                yf_calculated = euler_y[-1] if euler_y else y0
                self.record_solution_to_csv(self.record_counter,
self.ode_function_str.get(), "Euler", h, x0, y0,
                                            yf_calculated, end_time -
start_time)

            elif method_selection == "Improved Euler":
                start_time = time.time()
                improved_x, improved_y = self._improved_euler_method(x0,
y0, xf, h)
                end_time = time.time()
                if not improved_x: return

                all_methods_plot_data.append(("Improved Euler",
improved_x, improved_y))
                all_methods_table_data_combined = [("Improved Euler",
improved_x, improved_y)]

                yf_calculated = improved_y[-1] if improved_y else y0
                self.record_solution_to_csv(self.record_counter,
self.ode_function_str.get(), "Improved Euler", h, x0,
                                            y0, yf_calculated, end_time -
start_time)

            elif method_selection == "RK4":
                start_time = time.time()
                rk4_x, rk4_y = self._runge_kutta_4th_order(x0, y0, xf, h)
                end_time = time.time()
                if not rk4_x: return

                all_methods_plot_data.append(("Runge-Kutta (RK4)", rk4_x,
rk4_y))
                all_methods_table_data_combined = [("Runge-Kutta (RK4)",
rk4_x, rk4_y)]

                yf_calculated = rk4_y[-1] if rk4_y else y0
                self.record_solution_to_csv(self.record_counter,
self.ode_function_str.get(), "RK4", h, x0, y0,
                                            yf_calculated, end_time -
start_time)

            elif method_selection == "All":
                start_time_euler = time.time()
                euler_x, euler_y = self._euler_method(x0, y0, xf, h)
                end_time_euler = time.time()
                if not euler_x: return

                start_time_improved = time.time()
```

```
                improved_x, improved_y = self._improved_euler_method(x0,
y0, xf, h)
                end_time_improved = time.time()
                if not improved_x: return

                start_time_rk4 = time.time()
                rk4_x, rk4_y = self._runge_kutta_4th_order(x0, y0, xf, h)
                end_time_rk4 = time.time()
                if not rk4_x: return

                all_methods_plot_data.append(("Euler", euler_x, euler_y))
                all_methods_plot_data.append(("Improved Euler",
improved_x, improved_y))
                all_methods_plot_data.append(("Runge-Kutta (RK4)", rk4_x,
rk4_y))

                max_len = max(len(euler_x), len(improved_x), len(rk4_x))
                for i in range(max_len):
                    x_val = euler_x[i] if i < len(euler_x) else np.nan
                    y_euler = euler_y[i] if i < len(euler_y) else np.nan
                    y_improved = improved_y[i] if i < len(improved_y)
else np.nan
                    y_rk4 = rk4_y[i] if i < len(rk4_y) else np.nan
                    all_methods_table_data_combined.append((x_val,
y_euler, y_improved, y_rk4))

                yf_calculated_euler = euler_y[-1] if euler_y else y0
                self.record_solution_to_csv(self.record_counter,
self.ode_function_str.get(), "Euler", h, x0, y0,
                                            yf_calculated_euler,
end_time_euler - start_time_euler)

                yf_calculated_improved = improved_y[-1] if improved_y
else y0
                self.record_solution_to_csv(self.record_counter,
self.ode_function_str.get(), "Improved Euler", h, x0,
                                            y0, yf_calculated_improved,
end_time_improved - start_time_improved)

                yf_calculated_rk4 = rk4_y[-1] if rk4_y else y0
                self.record_solution_to_csv(self.record_counter,
self.ode_function_str.get(), "RK4", h, x0, y0,
                                            yf_calculated_rk4,
end_time_rk4 - start_time_rk4)

            else:
                messagebox.showerror("Error", "No method selected.")
                return

            if all_methods_plot_data:
                self.plot_results(all_methods_plot_data)
                self.display_table_results(method_selection,
all_methods_table_data_combined)
```

```
        except ValueError:
            messagebox.showerror("Input Error", "Please enter valid 
numerical values for x0, y0, xf, and h.")
        except Exception as e:
            messagebox.showerror("Error", f"An unexpected error occurred: 
{e}")

    def plot_results(self, methods_plot_data):
        self.ax.clear()
        markers = ['o', 's', '^', 'D', 'v', '>', '<', 'p', '*', 'h', 'H', 
'+', 'x', 'd', '|', '_']
        linestyles = ['-']
        colors = ['blue', 'orange', 'green', 'red', 'purple', 'brown', 
'pink', 'gray', 'olive', 'cyan']

        if methods_plot_data:
            for i, (label, x_vals, y_vals) in 
enumerate(methods_plot_data):
                if x_vals and y_vals:
                    marker = markers[i % len(markers)]
                    linestyle = linestyles[0]
                    color = colors[i % len(colors)]
                    self.ax.plot(x_vals, y_vals, marker=marker, 
linestyle=linestyle, color=color, label=label)

            self.ax.set_title(f"Solution for dy/dx = 
{self.ode_function_str.get()}")
            self.ax.set_xlabel("x")
            self.ax.set_ylabel("y")
            self.ax.legend()
            self.ax.grid(True)
        else:
            self.ax.set_title("No Solution to Plot")
            self.ax.set_xlabel("x")
            self.ax.set_ylabel("y")

        self.canvas.draw()

    def display_table_results(self, method_type, data_to_display):
        for item in self.results_tree.get_children():
            self.results_tree.delete(item)

        if method_type == "All":
            for x_val, y_euler, y_improved, y_rk4 in data_to_display:
                self.results_tree.insert("", "end", 
                                        values=(f"{x_val:.4f}", 
f"{y_euler:.4f}", f"{y_improved:.4f}", f"{y_rk4:.4f}"))
        else:
            if data_to_display:
                method_label, x_vals, y_vals = data_to_display[0]
                for i in range(len(x_vals)):
                    self.results_tree.insert("", "end", 
values=(f"{x_vals[i]:.4f}", f"{y_vals[i]:.4f}"))
```

```python
    def _clear_plot_and_table(self):
        self.x_values = []
        self.y_values = []
        self.ax.clear()
        self.canvas.draw()
        for item in self.results_tree.get_children():
            self.results_tree.delete(item)

    def clear_results(self):
        self.ode_function_str.set("")
        self.x0_str.set("")
        self.y0_str.set("")
        self.xf_str.set("")
        self.h_str.set("")
        self.selected_method.set("Euler")

        self._clear_plot_and_table()

    def record_solution_to_csv(self, run_id, ode_equation, method,
step_size, x0, y0, yf_calculated, time_taken):
        file_exists = os.path.isfile(self.csv_file_path)

        with open(self.csv_file_path, 'a', newline='') as csvfile:
            fieldnames = [
                'Run ID', 'ODE Equation', 'Numerical Method', 'Step Size
(h)',
                'Initial X (x0)', 'Initial Y (y0)', 'Calculated Y(xf)',
'Time Taken (s)'
            ]
            writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

            if not file_exists:
                writer.writeheader()

            writer.writerow({
                'Run ID': run_id,
                'ODE Equation': ode_equation,
                'Numerical Method': method,
                'Step Size (h)': f"{step_size:.4f}",
                'Initial X (x0)': f"{x0:.4f}",
                'Initial Y (y0)': f"{y0:.4f}",
                'Calculated Y(xf)': f"{yf_calculated:.4f}",
                'Time Taken (s)': f"{time_taken:.6f}"
            })


if __name__ == "__main__":
    root = tk.Tk()
    app = ODESolverApp(root)
    root.mainloop()
```