

Differential Equations programming assignment

Ramil Askarov, BS2-8
Fall 18 Innopolis University

Link to GitHub project: https://github.com/Rome314/DE_Assignment

Initial Value problem:

Problem:

$$y' = e^{-\sin x} - y \cdot \cos(x)$$

$$y(0) = 1$$

$$x \in [0:10]$$

Exact solution for IVP is:

$$y = e^{-\sin(x)} + x \cdot e^{-\sin(x)}$$

Code explanation:

main.py:

Just for running and change some constants, such as steps count or bound for calculating interval(X)

```
from DE_Assignment.plotting import *

# Costs:
# _____
steps_count = 100
X = 10

glob_err_start = 20
glob_err_end = 1000
# _____

# Plotting graphs for each task:
plot_methods(steps_count, X)
plot_local_error(steps_count, X)
plot_global_error(glob_err_start, glob_err_end, X)
```


functions.py:

Here is main formulas for function and numeric method. If will be necessary to change function, need

simply change formula in func **f** and **exact**.

```
from math import exp, sin, cos

# Original function and it's exact solution:
def f(x, y):
    return exp(-sin(x)) - y * cos(x)

def exact(x):
     return exp(-sin(x)) + x * exp(-sin(x))

# Helping fxunctions for Runge-Kutta method:

def k1(x, y):
    return f(x, y)

def k2(x, y, h):
    return f(x + h / 2, y + h * k1(x, y) / 2)

def k3(x, y, h):
    return f(x + h / 2, y + h * k2(x, y, h) / 2)

def k4(x, y, h):
    return f(x + h, y + h * k3(x, y, h))

def rk_delta(x, y, h):
    return h / 6 * (k1(x, y) + 2 * k2(x, y, h) + 2 * k3(x, y, h) + k4(x, y, h))

# Helping function for improved Euler method:

def imp_Euler_delta(x, y, h):
    return h * f(x + h / 2, y + h / 2 * f(x, y))
```

functions_calculator.py:

Module for calculating y values for each method. If function is changed and initial values too, have to change values in each def beginning, for example:

$x = [-5.0] \Rightarrow x = [1.0]$

$y = [2] \Rightarrow y = [0]$

```
def euler_solution(count_steps, final_x):
    # Initial values
    x = [0]
    y = [1]
    # step size
    h = (final_x - x[0]) / count_steps

    for i in range(count_steps):
        # here graph is created point by point
        x.append(x[i] + h)
        y.append(y[i] + h * (f(x[i], y[i])))

    return x, y

def exact_solution(steps_count, final_x):
    x = [0] # x0
    y = [1] # y0

    h = (final_x - x[0]) / steps_count

    for i in range(steps_count):
        x.append(x[i] + h)
        y.append(exact(x[i]))

    return x, y

def runge_kutta_solution(steps_count, final_x):
    x = [0] # x0
    y = [1] # y0

    h = (final_x - x[0]) / steps_count

    for i in range(steps_count):
        x.append(x[i] + h)
        y.append(y[i] + rk_delta(x[i], y[i], h))
    return x, y

def improved_euler_solution(steps_count, final_x):
    x = [0] # x0
    y = [1] # y0

    h = (final_x - x[0]) / steps_count

    for i in range(steps_count):
        x.append(x[i] + h)
        y.append(y[i] + imp_Euler_delta(x[i], y[i], h))
    return x, y
```

ploting.py:

Module where graphs are creating

```
from DE_Assigment.functions_calculator import *
from matplotlib import pyplot as plt
from math import fabs

#Plotting usual graphs for each ,ethod:
# @steps_count - accuracy value
# @x_final - end of plotting interval
def plot_methods(steps_count, x_final):
    x_euler, y_euler = euler_solution(steps_count, x_final)
    x_exact, y_exact = exact_solution(steps_count, x_final)
    x_euler_improved, y_euler_improved = improved_euler_solution(steps_count, x_final)
    x_runge_kutta, y_runge_kutta = runge_kutta_solution(steps_count, x_final)

    plt.title("All methods result")
    plt.plot(x_euler, y_euler, label="Euler method")
    plt.plot(x_exact, y_exact, label="Exact solution")
    plt.plot(x_euler_improved, y_euler_improved, label="Improved Euler method")
    plt.plot(x_runge_kutta, y_runge_kutta, label="Runge-Kutta's method")
    plt.ylabel("y")
    plt.xlabel("x")
    plt.legend()
    plt.show()

# Plotting local errors graphs for each method:
# @steps_count - accuracy value
# @x_final - end of plotting interval
def plot_local_error(steps_count, x_final):
    x_exact, y_exact = exact_solution(steps_count, x_final)
    x_euler_improved, y_euler_improved = improved_euler_solution(steps_count, x_final)
    x_euler, y_euler = euler_solution(steps_count, x_final)
    x_runge_kutta, y_runge_kutta = runge_kutta_solution(steps_count, x_final)

    runge_kutta_error = [0.0]
    euler_imp_error = [0.0]
    euler_error = [0.0]

    for i in range(steps_count):
        runge_kutta_error.append(fabs(y_exact[i] - y_runge_kutta[i]))
        euler_imp_error.append(fabs(y_exact[i] - y_euler_improved[i]))
        euler_error.append(fabs(y_exact[i] - y_euler[i]))

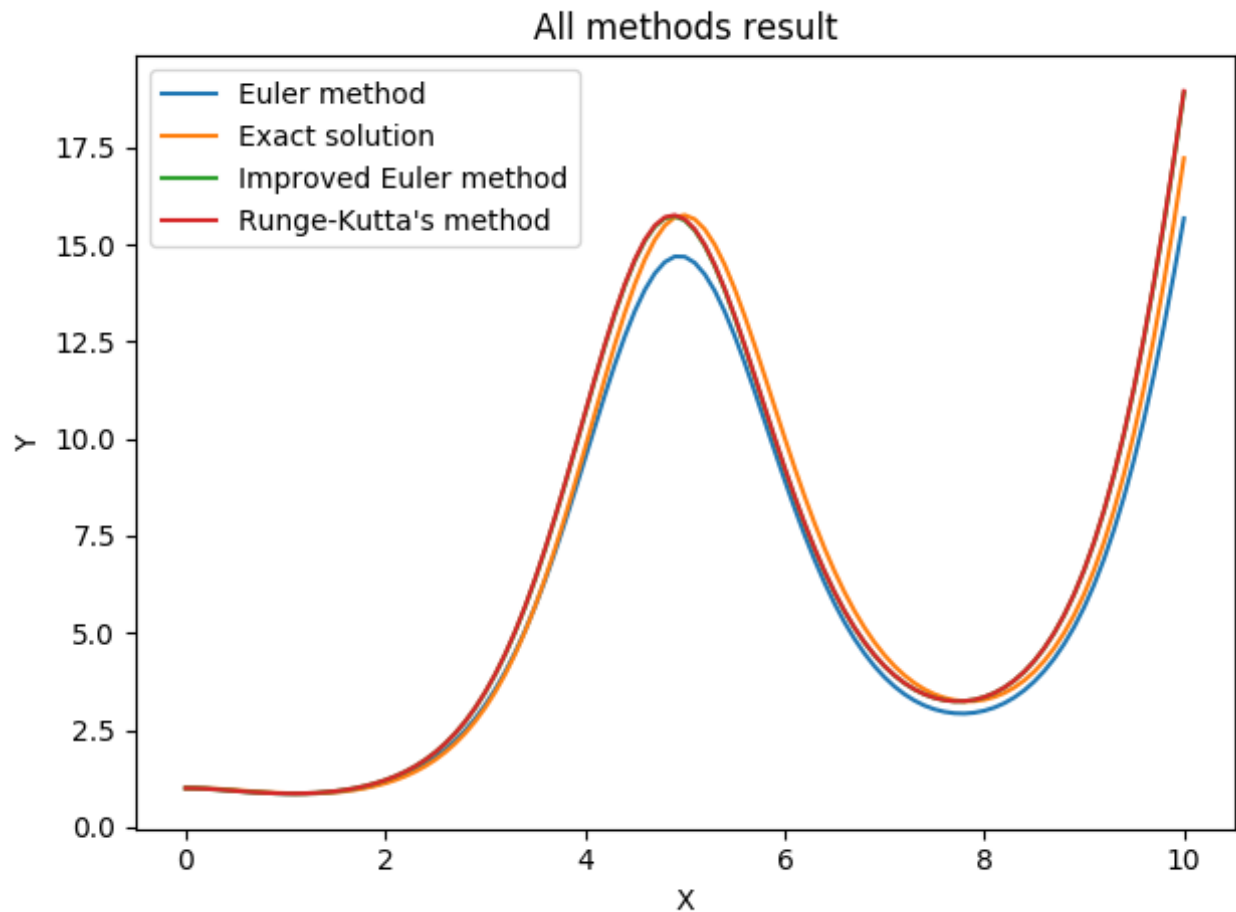
    plt.title("Local errors graph")
    plt.plot(x_euler, euler_error, label="Euler method")
    plt.plot(x_euler_improved, euler_imp_error, label="Improved Euler method")
    plt.plot(x_runge_kutta, runge_kutta_error, label="Runge-Kutta's method")
    plt.ylabel("Error")
    plt.xlabel("x")
    plt.legend()
    plt.show()
```

Program running results:

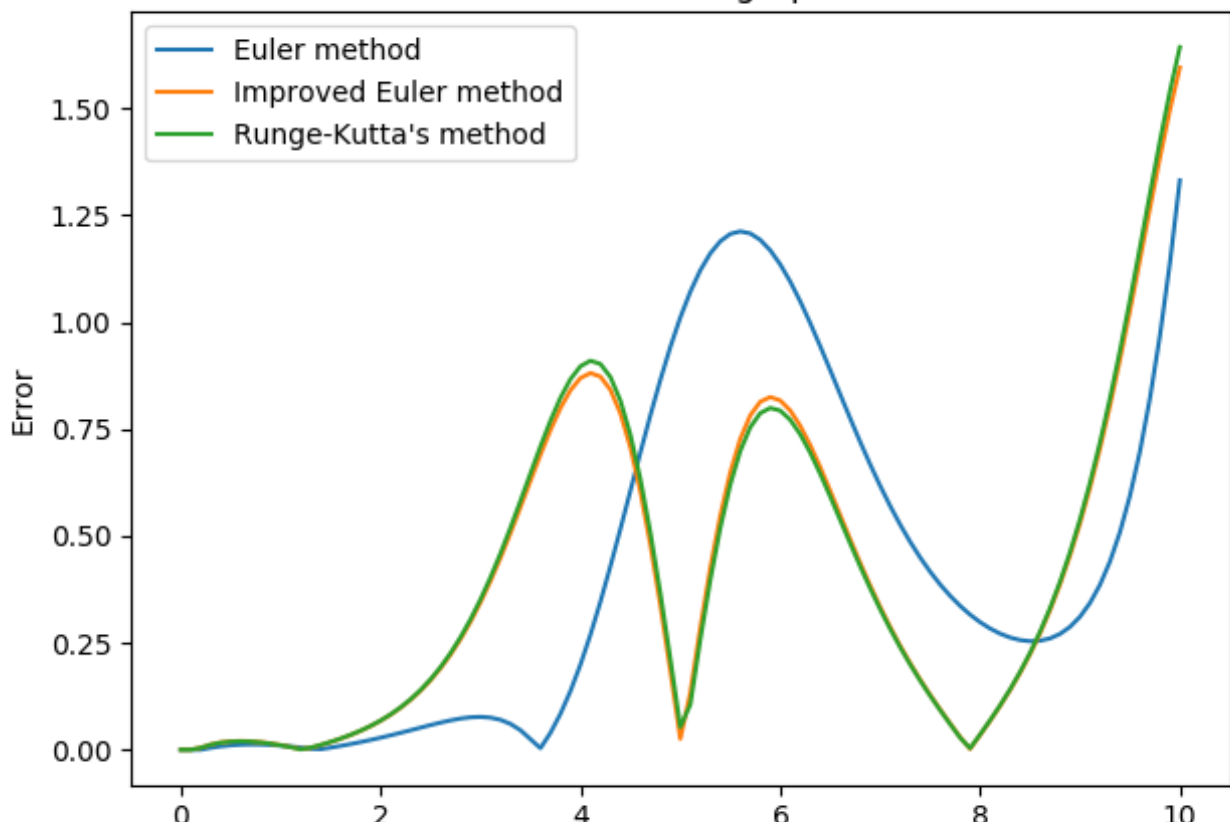
With iterations count = 1000

Interval : [0:1]

Interval for global error: [20:1000]



Local errors graph



Global errors graph

