Computational Practicum

Student: Arlan Kuralbayev

Mail: a.kuralbayev@innopolis.university

Group: Bs20-03

Variant: 5

Task

Given the IVP with ODE of first order on some interval

$$\begin{cases} y' = f(x,y) \\ y(x_0) = y_0 \\ x \in (x_0, X) \end{cases}$$

Given: $f(x,y) = y/x + x\cos(x);$ $x_0 = \pi;$ $y_0 = 1;$ $X = 4\pi$

$$\begin{cases} y' = y/x + xcos(x) \\ y(\pi) = 1 \\ x \in (\pi, 4\pi) \end{cases}$$

Exact solution:

$$y' = y/x + x\cos(x)$$

$$y' - y/x = x\cos(x)$$

Solve $y' + a(x)y = 0$ $\quad y' - y/x = 0$

Substitute $y' = \frac{dy}{dx}$ $\quad \frac{dy}{dx} = \frac{y}{x}$

$$\frac{1}{y}dy = \frac{1}{x}dx$$

Integral both sides $\quad \int \frac{1}{y}dy = \int \frac{1}{x}dx$

$$\ln|y| = \ln|x| + \ln|C|$$

Use $\log x + \log y = \log xy$ $\quad \ln|y| = ln|Cx|$

$$y = Cx$$

Now find particular solution $\quad y_p = C(x)x$

Derivate both sides $\quad y_p' = C'(x)x + C(x)$

Substitute $y' = y_p'$ $\quad C'(x)x + C(x) = y/x + x\cos(x)$

Substitute $y = Cx$ $\quad C'(x)x + \cancel{C(x)} = \cancel{\frac{C(x)x}{x}} + x\cos(x)$

$$C'(x)\cancel{x} = \cancel{x}\cos(x)$$

Integral both sides $\quad C(x) = \sin(x)$

$y_{exact} = y_g + y_p$ $\quad y = Cx + C(x)x$

Substitute $C(x) = \sin(x)$ $\quad y = Cx + \sin(x)x$

$$y = x(C + \sin(x))$$

Put initial value y_0, x_0 $\quad 1 = \pi C + \pi\sin(\pi)$

$$1 = \pi C$$

$C = \frac{1}{\pi}$

$C = \frac{y_0}{x_0} - \sin(x_0)$

Answer $$\boxed{y = x(\frac{1}{\pi} + \sin(x))}$$

How to work with the program:

- Just run "**Main.py**" in **Methods** folder
- Input all data to terminal
- Then all files will be generated at **Reports** folder

How program works:

in "**Main.py**" we interact with terminal, to enter Inputs, after that we create object of class *GUI* and pass our inputs, then we call *solve()* function, in *GUI* it will calculate all graphs and keep it, after that we call *show()* function which will generate our graphs in **.png** in **Report** folder

```
gui = Gui(x_start, y0, x_end, N, y_prime_formula, y_formula)
gui.solve()
gui.show()
```

in "**GUI.py**" we have 3 methods: *constructor*(*__init__*), *solve*, *show*. In *constructor* we create all arrays for graphs, and create objects of classes: **EulerMethod**, **ImprovedEulerMethod**, **RungeKuttaMethod**. In *show()* we plotting ready arrays to graphs and save it to **Report**. In *solve()* we have 2 *for* one for solving **Euler**, **Imporved Euler**, **Runge-Kutta methods**, **LTE**, **GTE**, another for **Errors depending on the number of grid cells (N)**.

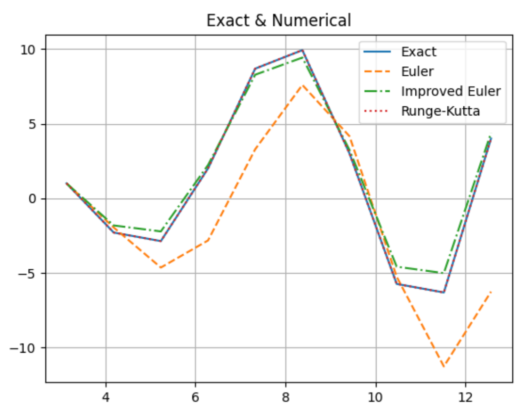
```
def solve(self):
        # Exact, Euler, Improved Euler, Runge-Kutta methods, LTE, GTE
        h = (self.x_end - self.x_start) / (self.N - 1)
        for i in range(1, self.N, 1):
            self.y_exact = np.append(self.y_exact, eval(self.y_formula, {"x": self.x[i], "np": np, "C": self.C}))
            self.y_euler = np.append(self.y_euler, self.euler_method.get_y(self.x[i - 1], self.y_euler[i - 1], h))
            self.y_imp_euler = np.append(self.y_imp_euler, self.imp_euler_method.get_y(self.x[i - 1], self.y_imp_euler[i - 1], h))
            self.y_runge_kutta = np.append(self.y_runge_kutta, self.runge_kutta_method.get_y(self.x[i - 1], self.y_runge_kutta[i - 1], h))
            self.lte_euler = np.append(self.lte_euler,
                abs(self.euler_method.get_y(self.x[i - 1], self.y_exact[i - 1], h) - self.y_exact[i]))
            self.lte_imp_euler = np.append(self.lte_imp_euler,
                abs(self.imp_euler_method.get_y(self.x[i - 1], self.y_exact[i - 1], h) - self.y_exact[i]))
            self.lte_runge_kutta = np.append(self.lte_runge_kutta,
                abs(self.runge_kutta_method.get_y(self.x[i - 1], self.y_exact[i - 1], h) - self.y_exact[i]))
            self.gte_euler = np.append(self.gte_euler, abs(self.y_exact[i] - self.y_euler[i]))
            self.gte_imp_euler = np.append(self.gte_imp_euler, abs(self.y_exact[i] - self.y_imp_euler[i]))
            self.gte_runge_kutta = np.append(self.gte_runge_kutta, abs(self.y_exact[i] - self.y_runge_kutta[i]))

        # Errors depending on number of points on graph (N)
        for n in range(2, 100, 1):
            h = (self.x_end - self.x_start) / (n - 1)
            x2 = np.arange(self.x_start, self.x_end + h / 2, h)
            temp_euler = temp_imp_euler = temp_runge_kutta =
            exact = euler = imp_euler = runge_kutta = self.y0
            for i in range(1, n, 1):
                exact = eval(self.y_formula, {"x": x2[i], "np": np, "C": self.C})
                euler = self.euler_method.get_y(x2[i - 1], euler, h)
                imp_euler = self.imp_euler_method.get_y(x2[i - 1], imp_euler, h)
                runge_kutta = self.runge_kutta_method.get_y(x2[i - 1], runge_kutta, h)
                temp_euler += abs(exact - euler)
                temp_imp_euler += abs(exact - imp_euler)
                temp_runge_kutta += abs(exact - runge_kutta)
            self.sum_euler = np.append(self.sum_euler, temp_euler)
            self.sum_imp_euler = np.append(self.sum_imp_euler, temp_imp_euler)
            self.sum_runge_kutta = np.append(self.sum_runge_kutta, temp_runge_kutta)
```

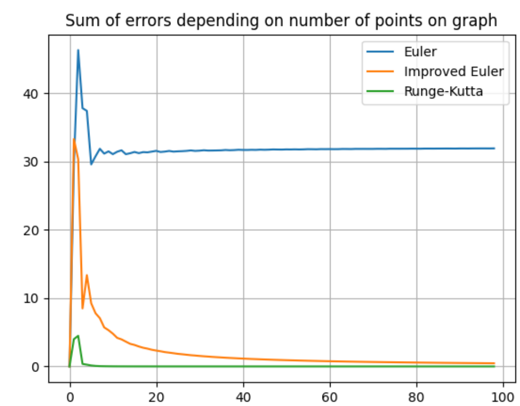
in "**EulerMethod.py**", "**ImprovedEulerMethod.py**", "**RungeKuttaMethod.py**" in each we have 2 variables: *h, y_prime_formula* for calculating these methods. And 3 methods: *constructor(__init__), get_y_prime(x, y), get_y(x, y, h)*. In *get_y_prime(x, y)* we will put ***x*** and ***y*** to ***y' formula*** and return result. In *get_y(x, y, h)* we will solve Euler, Improved Euler, Runge-Kutta method

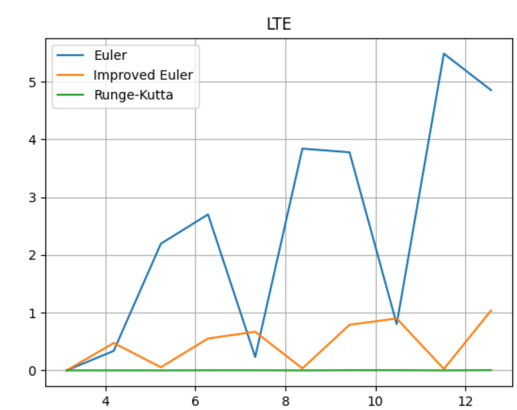
```
class RungeKuttaMethod:
    def __init__(self, h, y_prime_formula):
        self.h = h
        self.y_prime_formula = y_prime_formula

    def get_y_prime(self, x, y):
        return eval(self.y_prime_formula, {"x": x, "y": y, "np": np})

    def get_y(self, x_prev, y_prev, h):
        k1 = self.get_y_prime(x_prev, y_prev)
        k2 = self.get_y_prime(x_prev + h / 2, y_prev + h * k1 / 2)
        k3 = self.get_y_prime(x_prev + h / 2, y_prev + h * k2 / 2)
        k4 = self.get_y_prime(x_prev + h, y_prev + h * k3)
        return y_prev + h / 6 * (k1 + 2 * k2 + 2 * k3 + k4)
```

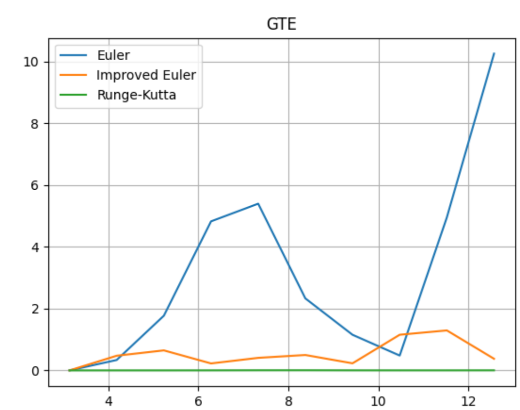
Graphs:

for N = 10

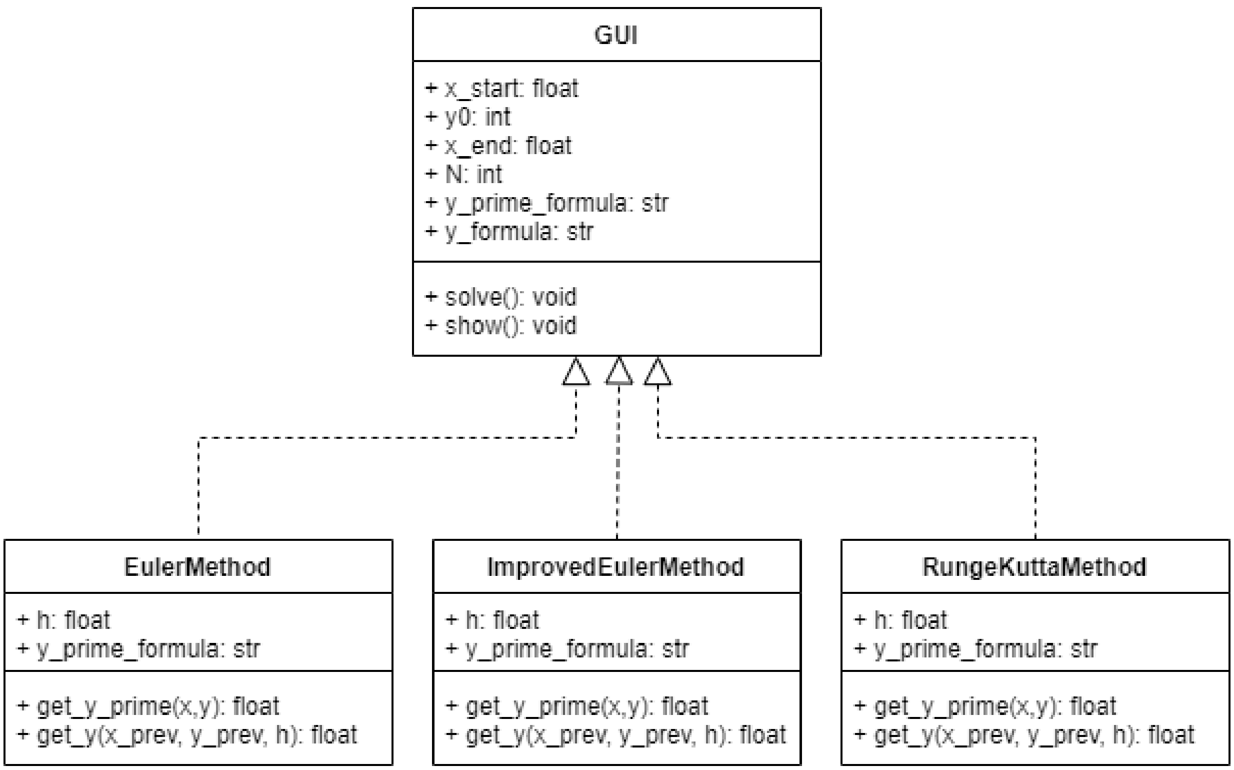








UML:


GUI
+ x_start: float
+ y0: int
+ x_end: float
+ N: int
+ y_prime_formula: str
+ y_formula: str
+ solve(): void
+ show(): void
EulerMethod
+ h: float
+ y_prime_formula: str
+ get_y_prime(x,y): float
+ get_y(x_prev, y_prev, h): float
ImprovedEulerMethod
+ h: float
+ y_prime_formula: str
+ get_y_prime(x,y): float
+ get_y(x_prev, y_prev, h): float
RungeKuttaMethod
+ h: float
+ y_prime_formula: str
+ get_y_prime(x,y): float
+ get_y(x_prev, y_prev, h): float

Conclusion:

In **Sum of errors** we can see that the more points we use, the less total errors.

Also after comparing all methods, we can note that **Runge-Kutta** method is the most accurate to exact.