Assignment Report

Marina Smirnova BS18-06

November 2019

Contents

1 Analytical Solution **2**

2 Graphs of Solutions **2**

3 Graphs of Errors **4**

4 OOP Design **5**

5 Some Interesting Parts of Source Code **6**

6 UML Diagram **6**

1 Analytical Solution

The given equation and IVP $\begin{cases} \frac{y}{x} - y - x = y' \\ y(1) = 0 \\ x \in [1; 10] \end{cases}$

•This is Linear Non-homogeneous equation. By conditions of the equation $x \in (-\infty; 0) \cup (0; -\infty)$. Firstly, it's needed to solve complementary equation and find y_1. Then use substitution $y = y_1 * u$. And find u by the fact $u' = \frac{f(x)}{y_1}$.

- Complementary equation:

$y_1' + y_1 * \left(1 - \frac{1}{x}\right) = 0$
$\int \frac{1}{y_1} dy_1 = \int \left(1 - \frac{1}{x}\right) dx$
$\ln|y_1| = -x + \ln|x|$
$y_1 = \frac{x}{e^x}$

- Substitution: $y = y_1 * u$

$u' = \frac{f(x)}{y_1}$
$u' = -e^x$
$u = -e^x + C$

- Answer Exact Solution: $y = -x + \frac{x}{e^x} * C$

Range of Validity: $x \in (-\infty; 0) \cup (0; -\infty)$

- Solve IVP:

$0 = -1\frac{1}{e} * C$
$C = e$
$y = y_1 * u$

- Answer IVP: $y = -x + \frac{x}{e^x} * e$

2 Graphs of Solutions

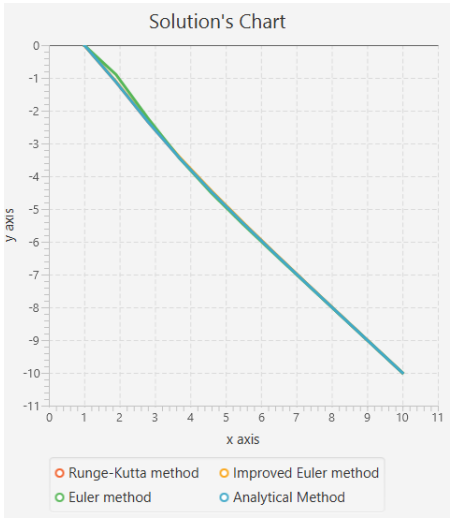
In this section, I would like to provide solutions for each method used (Exact, Euler, Improved Euler, Runge-Kutta).
The initial conditions for calculations:

$$\begin{cases} y = -x + \frac{x}{e^x} * e \\ y(1) = 0 \\ x \in [1; 10] \\ N = 10 => h = 0.9 \end{cases}$$

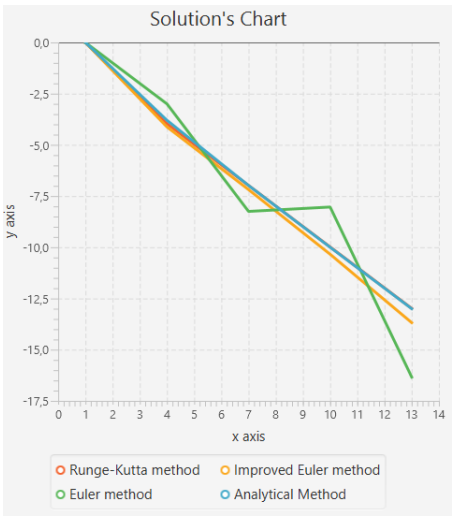
Figure 1: Table of results for each method.

	A	B	C	D	E
1	x0	Exact	Euler	Improved Euler	Runge-Kutta
2	1	0	0	0	0
3	1,9	-1,127517646	-0,9	-1,113157895	-1,127182974
4	2,8	-2,337163113	-2,226315789	-2,311462752	-2,336817089
5	3,7	-3,451339603	-3,458233083	-3,420397306	-3,450855425
6	4,6	-4,474310877	-4,517015139	-4,446011527	-4,473798911
7	5,5	-5,438900519	-5,475465346	-5,417004474	-5,438471146
8	6,4	-6,371093882	-6,393531773	-6,355869722	-6,370784670
9	7,3	-7,286594975	-7,298443583	-7,276749332	-7,286394130
10	8,2	-8,193877996	-8,199652471	-8,187830183	-8,193756763
11	9,1	-9,097237794	-9,099927104	-9,093661225	-9,097168487
12	10	-9,998765902	-9,9999855	-9,996710837	-9,998727900

Figure 2: Plots for all methods.



(a) N = 10



(b) N = 3

- **Analysis of plots:**

Although, for large quantity of N (even for N = 10) all plots seem to merge into one (close to the exact solution), it is possible to distinguish the difference among them using N = 3, for example. The most precise graph is done using the Runge-Kutta method, and the most rough precision is produced by Euler's method.

•The formulas common for all methods, valid for the case when $x_0 < x_n$:
$h = \frac{x_n - x_0}{N}$, where N is amount of steps.
$f(x, y) = \frac{y}{x} - y - x$

But if $x_0 > x_n$, we should use:
$h = \frac{x_0 - x_n}{N}$
$g(z, x) = -f(-x, y) = \frac{z}{x} + z - x$
Then values $z_1, z_2, ...z_n$ for $x_1, x_2, ...x_n$ correspond to values $y_1, y_2, ...y_n$ for the same values of x. So, instead of f(x_i, y_i) substitute g(x_i, z_i) in the next formulas.

- **Euler Method**

$x_{i+1} = x_i + h$
$y_{i+1} = y_i + h * f(x_i, y_i)$

- **Improved Euler Method**

$K_1 = f(x_i, y_i)$
$K_2 = f(x_i + h, y_i + h * K_1)$
$x_{i+1} = x_i + h$
$y_{i+1} = y_i + \frac{h}{2} * (K_1 + K_2)$

- **Runge-Kutta Method**

$K_1 = f(x_i, y_i)$
$K_2 = f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2} * K_1\right)$
$K_3 = f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2} * K_2\right)$
$K_4 = f(x_i + h, y_i + h * K_3)$
$x_{i+1} = x_i + h$
$y_{i+1} = y_i + \frac{h}{6} * (K_1 + 2 * K_2 + 2 * K_3 + K_4)$

3 Graphs of Errors

The approach for finding local and global errors is the same for all methods.
The formula for local errors is: $e_{i+1} = e_{global\ i+1} - e_{global\ i}$
The formula for global errors is: $e_{i+1} = y(x_{i+1}) - y_{i+1}$, where y_i is approximate value of y and $y(x_i)$ is exact value.
The magnitude of the local error is determined by the $y^{(n)}$ n-th derivative of the solution of the initial value problem. So, for Euler method it is $y^{(2)}$, for Improved Euler $y^{(3)}$, and for Runge-Kutta $y^{(5)}$. The local error with the Euler method is $O(h^2)$, with Improved Euler's method $O(h^3)$, and with Runge-Kutta $O(h^5)$.
The rate of change for global errors is is less on one power of h for each method. The global error with the Euler method is $O(h)$, with Improved Euler's method $O(h^2)$, and with Runge-Kutta $O(h^4)$.

Figure 3: Plots of Global Errors for N = 10.

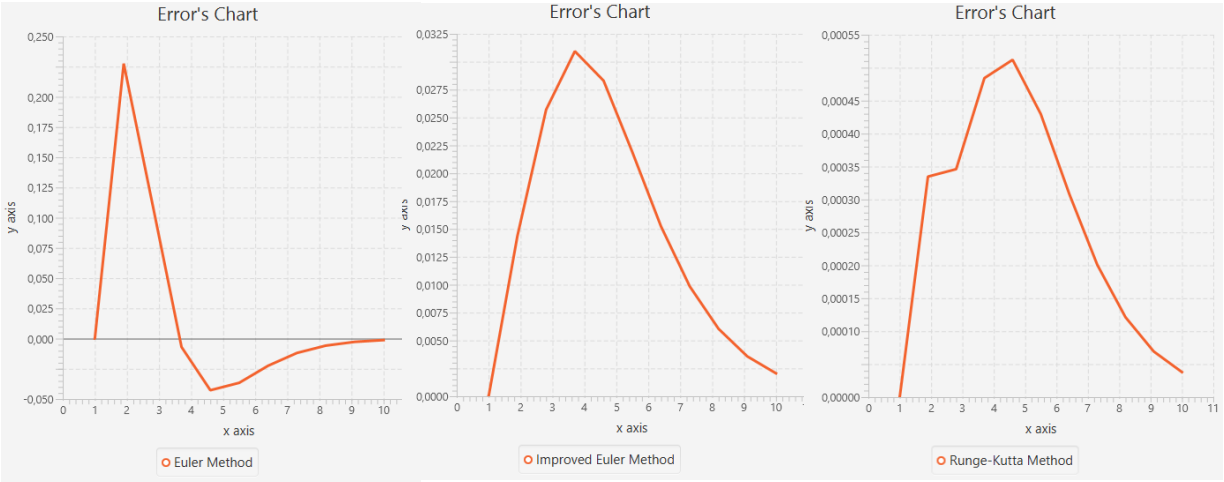
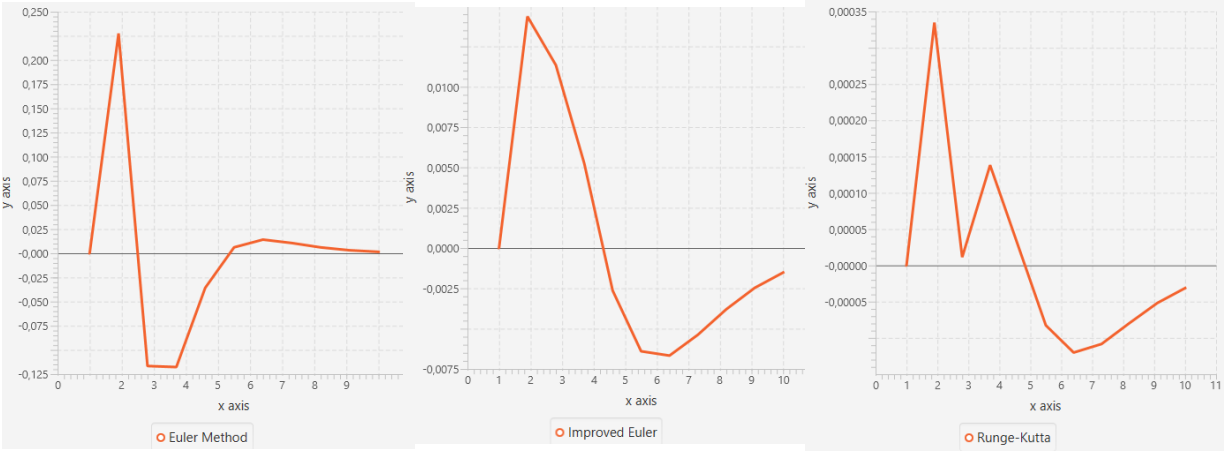


Figure 4: Plots of Local Errors for N = 10.



4 OOP Design

The programming language used for implementation of the assignment is Java (JavaFX).

SOLID principles:

•Single Responsibility Principle

"One class should have one and only one responsibility."

There are nine classes in my program and each has its own single reason of existence:
1. Main - to launch the application;
2. ControllerFXML - to react on user's interaction with the application;
3. MyChart (extends LineChart<Number, Number>) - to build graphs and specify particular features of graphs;
4. MyData - to store input data (initial value conditions) and retrieve it;
5. Method - to solve the equation and store necessary information;
6. AnalyticalMethod (extends Method) - to calculate value of y using Analytical Method;
7. EulerMethod (extends Method) - to calculate value of y using Euler Method;
8. ImprovedEuler (extends Method) - to calculate value of y using Improved Euler Method;
9. Runge-Kutta (extends Method) - to calculate value of y using Runge-Kutta Method.

•Open Closed Principle

"Software components should be open for extension, but closed for modification."

We can see how this principle is applied with the class Method and its children. The Method has a procedure solve() that repeatedly fill Series data structure with pairs (x, y) calculated within a loop. But the actual calculations of y value is done by function calculateY(double x, double y), which is overidden by each children to correspond with Analytical, Euler, Improved Euler and Runge-Kutta Methods.

• Liskov's Substitution Principle

"Derived types must be completely substitutable for their base types."

Implementation of the principle is noticeable when the program decides which method of solving to use getting as an input a name of the method. Then the object method of class Method is assigned to a particular subclass (with required solution approach) during runtime. So, objects of derived classes can fully substitute objects of corresponding superclasses.

• Interface Segregation Principle

"Clients should not be forced to implement unnecessary methods which they will not use".

There is a single interface - MethodInterface - in the program. It consists of three functions and two procedures:

1. function_f(double x_i, double y_i) - where f() is given by ODE y' = f() ;
2. exactSolution(double x_i) - to get y value using Analytical solution (also needed in calculations of a local error);
3. calculateY(double x, double y) - to find value of y using a particular method;
4. addData(double x, double y, double e) - insert data in Series data structures (series for solution and errors);
5. solve(MyData input) - to combine previous operations in a single loop and get final results.

And all of them are necessary to include within a single interface, because of their tight connection with each other. Also they all will be definitely used if a user want to solve the equation.

•Dependency Inversion Principle

"Depend on abstractions, not on concretions".

This principle is used for efficient implementation of a constructor for the class Method where it's given an object of type MyData with all needed input data. Also, methods in the class MyChart: init() is constructed in such a way that allows to optionally specify a chart's size and label, and plot() provides an opportunity to choose the type of graph (solution or error) and method of solving.
Consequently, the certain level of independence from concretions is achieved.

5 Some Interesting Parts of Source Code

Reference to source code on GitHub

- Range of Validity Constraints:

The only constraint on values of x and y: $x \in (-\infty; 0) \cup (0; -\infty)$. That's why there is a special filter for input fields of x_0 (value for IVP) and x_n (the interval of calculations is $[x_0; x_n]$) that doesn't allow to insert 0.
If during runtime calculations $x_i = 0$ appears, then this value is skipped and y_i is not calculated.

- Input Checking:

Overall checking of input for fields x_0, x_n, y_0, N is implemented using Event Filters: numericalFilter() is for y_0, N and xFilter() for values of x. Both of them check for valid symbols (digits, "." for floating point's numbers and "-" for negative values), otherwise it's impossible to proceed the program - all invalid characters are "consumed". But xFilter also inspects for 0 value inserted. And if 0 appears in x_0 or x_n, a new window with warning appears.

Figure 5: Checker for input's validity.

```
private EventHandler<KeyEvent> numericalFilter(){

    EventHandler<KeyEvent> eventHandler = new EventHandler<KeyEvent>() {
        public void handle(KeyEvent keyEvent) {
            if (!"0123456789.-".contains(keyEvent.getCharacter())) {
                keyEvent.consume();
            }
        }
    };
    return eventHandler;
}
```

- Full Screen option:

There two additional buttons which allow to build plots of solution and local errors, and display the result on the full screen for better comparison.

- Comparison of border's values:

There are two cases which must be treated a little differently: $x_0 < x_n$ and $x_0 > x_n$. That's why class Method contains boolean variable isReverse (false for the first case, true for the second).

Figure 6: Implementation.

```
public double function_f(double x_i, double y_i){
    if (!isReverse){
        return (y_i/x_i - y_i - x_i);
    }
    return (y_i/x_i + y_i - x_i);
}

public double exactSolution(double x_i){
    if (!isReverse){
        return -x_i+x_i/Math.exp(x_i)*c_const;
    }
    return x_i+x_i/Math.exp(x_i)*c_const;
}
```

6 UML Diagram

Figure 7: UML Diagram with public methods.

Figure 8: UML Diagram with fields.