DE: Computational Practicum (Var. 2)

Due on November 17, 2019 at 11:59pm

Nikolay Shilov

Artem Bahanov (B18-03)

Contents

Part 1. Exact solution

$$y'(x) = \frac{y}{x} - xe^{\frac{y}{x}},\ x_0 = 1,\ y_0 = 0 \tag{1}$$

Let us solve the given IVP analytically. We say that $x \neq 0$.

Solution. Let us make a substitution $u(x) = \frac{y}{x}$. Then $y = ux$ and $y' = u'x + u$. Let us substitute this to (1):

$$u'x + u = u - xe^u \tag{2}$$

Subtracting u from both parts and then dividing them by x yields:

$$u' = -e^u \tag{3}$$

We have separable equation.

$$u'e^{-u} = -1 \Leftrightarrow \int e^{-u}du = -\int dx \Leftrightarrow e^{-u} = x + C \Leftrightarrow u = -\ln(x + C),\ C \in \mathbb{R} \tag{4}$$

After substituting back we get: $y = ux = -x\ln(x + C)$, where C is some real constant. It is the most general solution. Let us find the solution of the IVP. From (4):

$$C = e^{-\frac{y}{x}} - x \tag{5}$$

Then $C(x_0, y_0) = 1 - 1 = 0$. The solution of the IVP:

$$y = -x\ln x \tag{6}$$

$\blacksquare$

Part 2. Design Overview

The application was developed for the Android mobile system. This system has been chosen since Android has better UX for such applications. For example, it is easier to scale graphs on Android device using multi-touch than it is on PC.

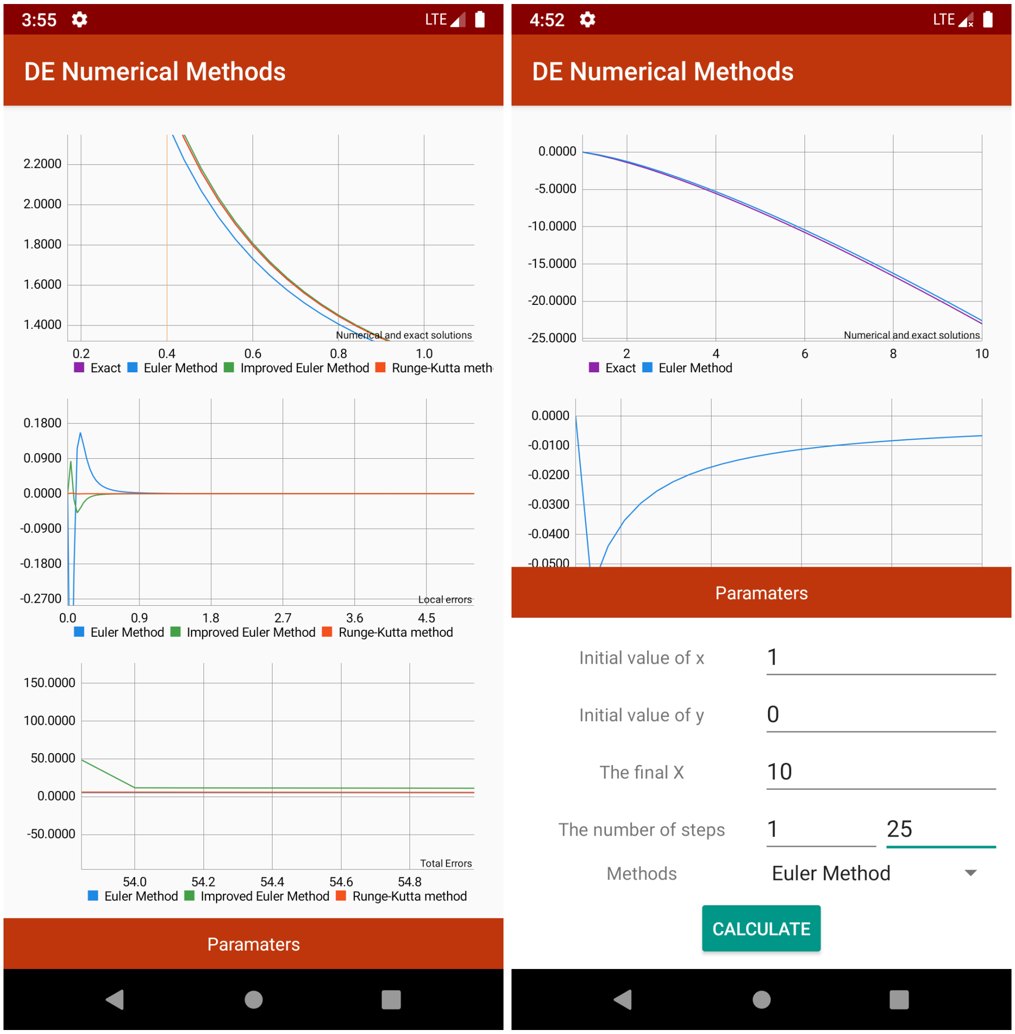
Before starting working on this project, I needed to choose how my application was going to draw graphs. Since I had not enough time to develop my own plot library, I had to choose an external library to use: MPAndroidChart, AnyChart, AndroidCharts, achartengine. I decided to use MPAndroidChart, due to a lot of benefits, including scaling on axes, highlighting values, etc.

First of all, the appropriate GUI was developed. It was decided to put all the plots on a single screen. Usually, Android devices are small, so there is no possibility to put a lot of information on a single screen, hence plots were merged and parameters were moved to the bottom sheet that can be easily opened by tapping or dragging the upper part of the sheet up.

The next step was to develop the the structure of classes, which is easy to extend and reuse.

First of all, I decided to create a class `Equation` that has several properties: `function: (x: Double, y: Double) -> Double` - function y' = f(x, y); `solution: (x: Double, c: Double) -> Double` - exact solution of the equation y = f(x, c), where c is some constant; `const: (x: Double, y: Double) -> Double` - function, which computes constant for given initial values; `includedPointsX` and `includedPointsY`

Part 2 continued on next page...



(a) Main Activity layout. (b) Opened bottom sheet with parameters.

Figure 1: The app design.

are predicates that determine x and y domain of the equation; `includedPointsDescription` is a string that is used for showing errors.

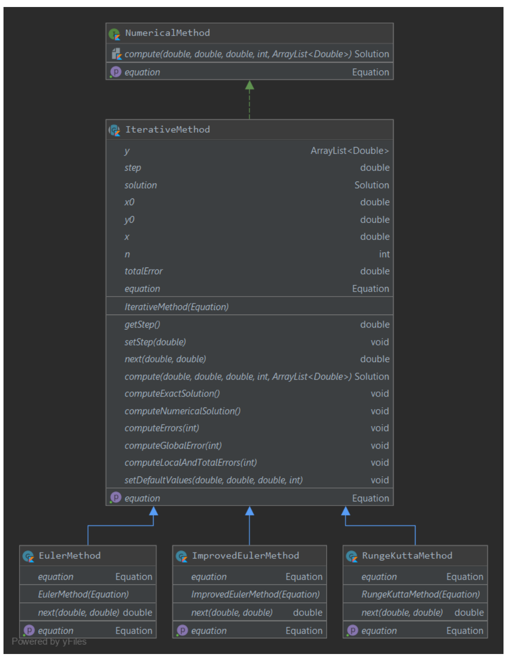
Secondly, I created an interface, called `NumericalMethod`, that has only one method `compute(x0:Double, y0:Double, x:Double, n:Int, exactSolution:ArrayList<Double>=null)`, where `x0`, `y0` - initial values, `x` - the final x value, `n` - the number of steps for numerical method on interval $[x0; x]$. `exactSolution` The interface represents generalization of all numerical methods.

Abstract class `Iterative method` implements `NumericalMethod` and unifies all the iterative methods and new abstract method (function) `next(xi:Double, yi:Double)`, which calculates the values on the next iteration of method based on some values `xi` and `yi`. These values can be either values of numerical solution or the values of exact solution on preceding step. The class has an implementation of `compute(...)`, since all the iterative methods are similar and `next(...)` is the only function that depends on a particular algorithm. It also has a primary constructor (Kotlin) and the field `equation: Equation`. Other methods that are implemented are private and not interesting, for example, `computeExactSolution()` computes exact solution.

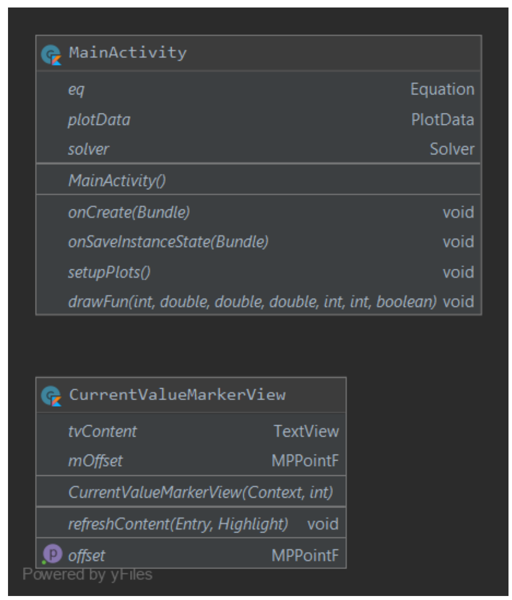
Classes `EulerMethod`, `ImprovedEulerMethod`, `RungeKuttaMethod` are subclasses of `IterativeMethod`. They are all implementations of a corresponding numerical method. The only function that is overridden is `next(...)`.

The last step was creating classes for exceptions. `NMException` is a super class for all of them. `NMArgumentException` is thrown when parameters are invalid (for example, `n > 0` should be held whenever `compute(...)` is called. `NMDomainException` is thrown when x or y value is not in the domain of equation. `NMStabilityException` is (rarely) thrown when a method seems to be unstable for a given initial value problem.

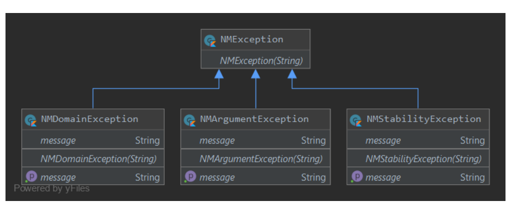
Class `Solver` was created since I needed to create some class that computes solutions for chosen methods. It has the only public function `generateSolutionPlotData(...)` that returns an object of type `PlotData` that is easily used for drawing plots. Also several auxiliary classes were created. All of them are drawn on the UML class diagram below.



(a) Package `numericalMethods.methodsDE`



(b) Auxiliary classes



(c) Package `numericalMethods.exception`

Figure 2: UML class diagram

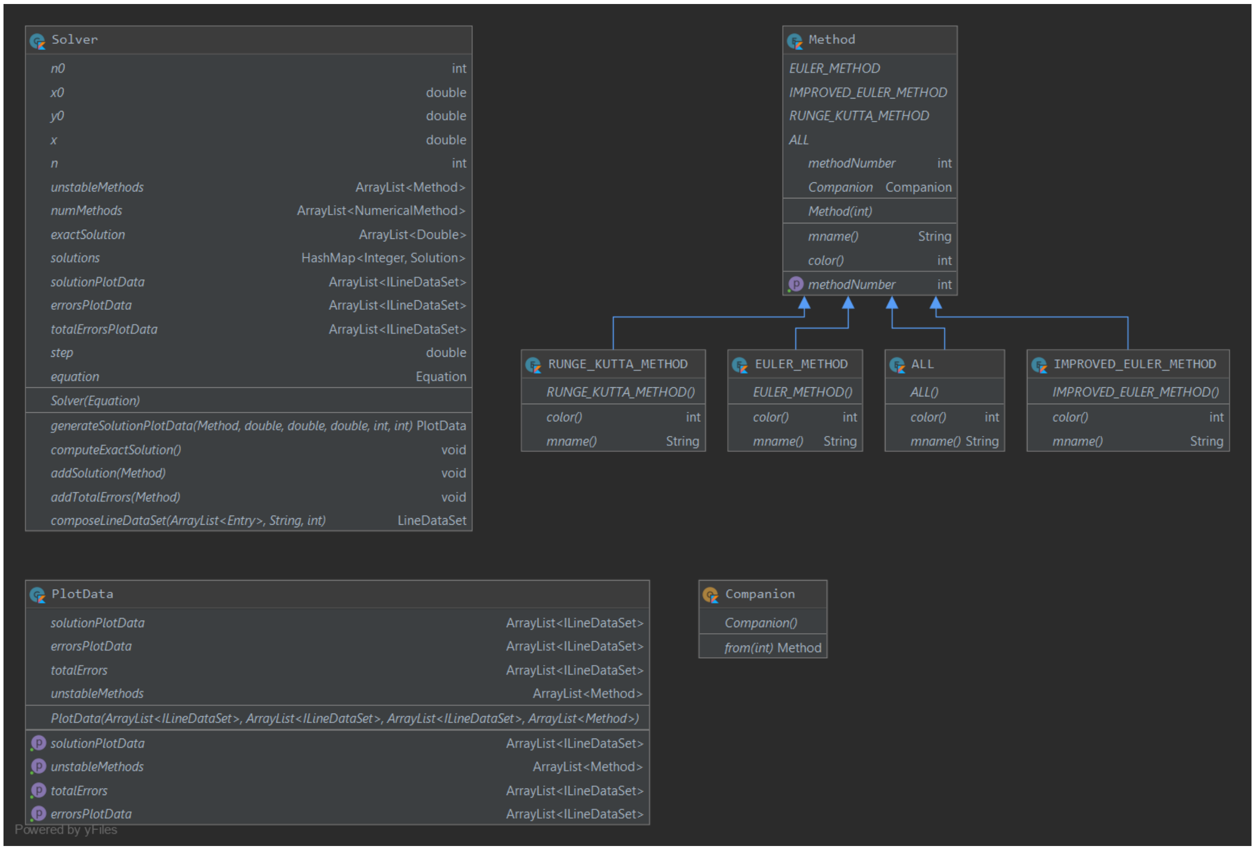


Figure 3: UML class diagram

Part 3. OOP Principles

The code is written on Kotling language (that is an OOP language) and satisfies SOLID principles.

1. **Single responsibility principle**
   All the classes are responsible for exactly one task. For example, `Equation` is responsible for storing information about differential equation; `RungeKuttaMethod` - for Runge-Kutta method computations; `MainActivity` - for UI part (including plots), etc.

2. **Open/closed principle**
   The code of my application is easy to extend without modifying it. For example, if one wants to create a new class, responsible for solving another numerical method, he/she can use `IterativeMethod` or `NumericalMethod` as base classes.

3. **Liskov substitution principle**
   Objects in my application can be substituted with their subtypes.

4. **Interface segregation principle**
   There is only one interface in my code (`NumericalMethod`) but it is client specific.

5. **Dependency inversion principle**
   In my code abstractions do not depend on details.

Part 4. Code

Listing 1: `NumericalMethod` interface.

```
interface NumericalMethod {
    val equation: Equation
        fun compute(
            x0: Double,
            y0: Double,
            x: Double,
            n: Int,
            exactSolution: ArrayList<Double>? = null
        ): Solution
}
```

Listing 2: Implementation of function compute inside `IterativeMethod`

```
abstract class IterativeMethod(override val equation: Equation) : NumericalMethod {
    ...

    protected abstract fun next(xi: Double, yi: Double): Double

    ...
    /**
     * Computes the solution numerically.
     *
     * @param x0 initial x value
     * @param y0 initial y value
     * @param x x value where the method finishes
     * @param n the number of steps
     * @param exactSolution predefined exact solution. Its size should be equal to n + 1
     *
     * @throws NMArgumentException if x0 >= x0 or n <= 0 or the size of exactSolution != n + 1
     */
    override fun compute(
        x0: Double,
        y0: Double,
        x: Double,
        n: Int,
        exactSolution: ArrayList<Double>?
    ): Solution {
        if (x0 >= x) throw NMArgumentException(
            "The initial x value should be less than the final one."
        )
        if (n <= 0) throw NMArgumentException(
            "The number of steps should be greater than 0"
        )
        setDefaultValues(x0, y0, x, n)

        if (exactSolution == null)
            computeExactSolution()
        else
            solution.exactSolution = exactSolution

        if (solution.exactSolution.size != n + 1)
            throw NMArgumentException("Exact solution size does not correspond to the number of points")

        computeNumericalSolution()

        solution.totalError = totalError

        return solution
    }

    ...

}
```

Part 4 continued on next page. . .

Listing 3: Implementation of method next in `EulerMethod`

```
class EulerMethod(override val equation: Equation) : IterativeMethod(equation) {
    override fun next(xi: Double, yi: Double): Double {
        return yi + step * equation.function(xi, yi)
    }
}
```

Listing 4: Implementation of method next in `ImprovedEulerMethod`

```
class ImprovedEulerMethod(override val equation: Equation) : IterativeMethod(equation){
    override fun next(xi: Double, yi: Double): Double {
        val f = equation.function
        val k1i = f(xi, yi)
        val k2i = f(xi + step, yi + step * k1i)
        return yi + step / 2 * (k1i + k2i)
    }
}
```

Listing 5: Implementation of method next in `RungeKuttaMethod`

```
class RungeKuttaMethod(override val equation: Equation) : IterativeMethod(equation) {
    override fun next(xi: Double, yi: Double): Double {
        val f = equation.function
        val k1i = f(xi, yi)
        val k2i = f(xi + step / 2, yi + step / 2 * k1i)
        val k3i = f(xi + step / 2, yi + step / 2 * k2i)
        val k4i = f(xi + step, yi + step * k3i)

        return yi + step / 6 * (k1i + 2 * k2i + 2 * k3i + k4i)
    }
}
```

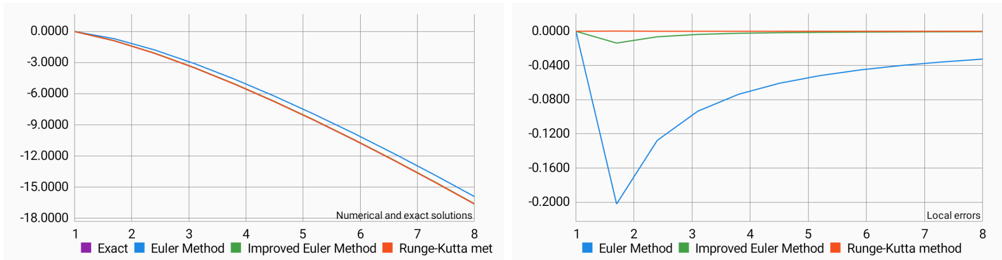
Listing 6: `Equation`

```
/**
 * This class represents the equation y' = f(x, y) without initial values.
 *
 * @property function y' = f(x, y)
 * @property solution the solution of the differential equation. c is not calculated (param)
 * @property const function that calculates constant for the solution, depending on initial values
 * @property includedPointsX predicate showing whether the points is in the x-domain
 * @property includedPointsY predicate showing whether the points is in the y-domain
 * @property includedPointsDescription text information about the domain of the DE
 */
class Equation(val function: (x: Double, y: Double) -> Double,
               val solution: (x: Double, c: Double) -> Double,
               val const: (x: Double, y: Double) -> Double,
               val includedPointsX: (x: Double, c: Double) -> Boolean,
               val includedPointsY: (y: Double) -> Boolean,
               val includedPointsDescription: String = "x and y should be in the domain")
```

The full source code is available on GitHub.

Part 5. Plot analysis

As you can see on the plots below, Euler method is the most inaccurate method (its maximum absolute value of local error is ≈ 0.2). As opposed to it Runge-Kutta method has the smallest local error (≈ 0.02). As it is supposed to be, the greater number of steps we choose the more accurate the methods are. Obviously, Euler Method has the smallest descent of total error.
This plots show that the total error is $O(h^2)$ for Euler Method, $O(h^3)$ for improved Euler method, and $O(h^4)$ for Runge-Kutta method.



(a) Exact and numerical solutions plot. (b) Local error plot for each numerical method.

Figure 4: $x_0 = 1$, $y_0 = 0$, $X = 8$, $N = 10$

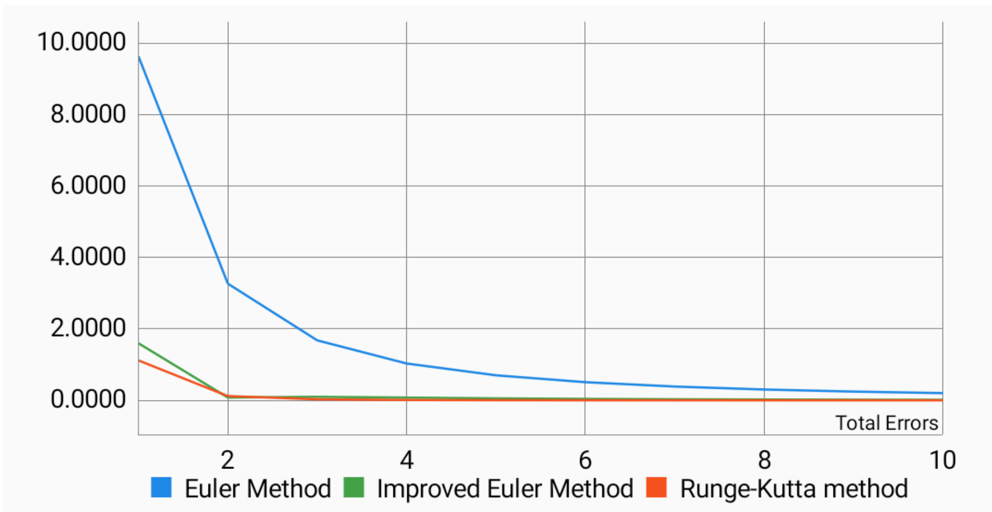


Figure 5: Total errors graph on $N \in [1, 10]$

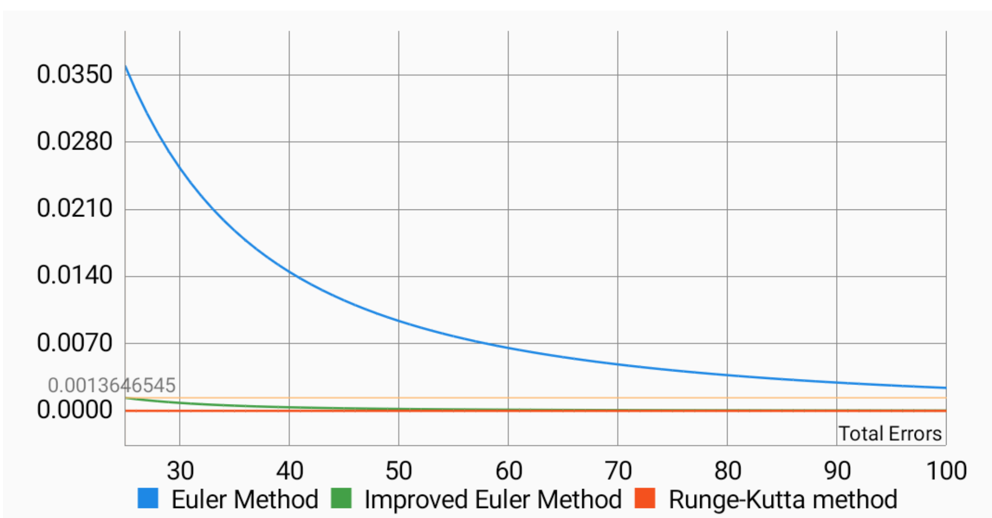


Figure 6: Total errors graph on $N \in [25, 100]$