Differential Equations

Computational Assignment

Alfiya Mussabekova

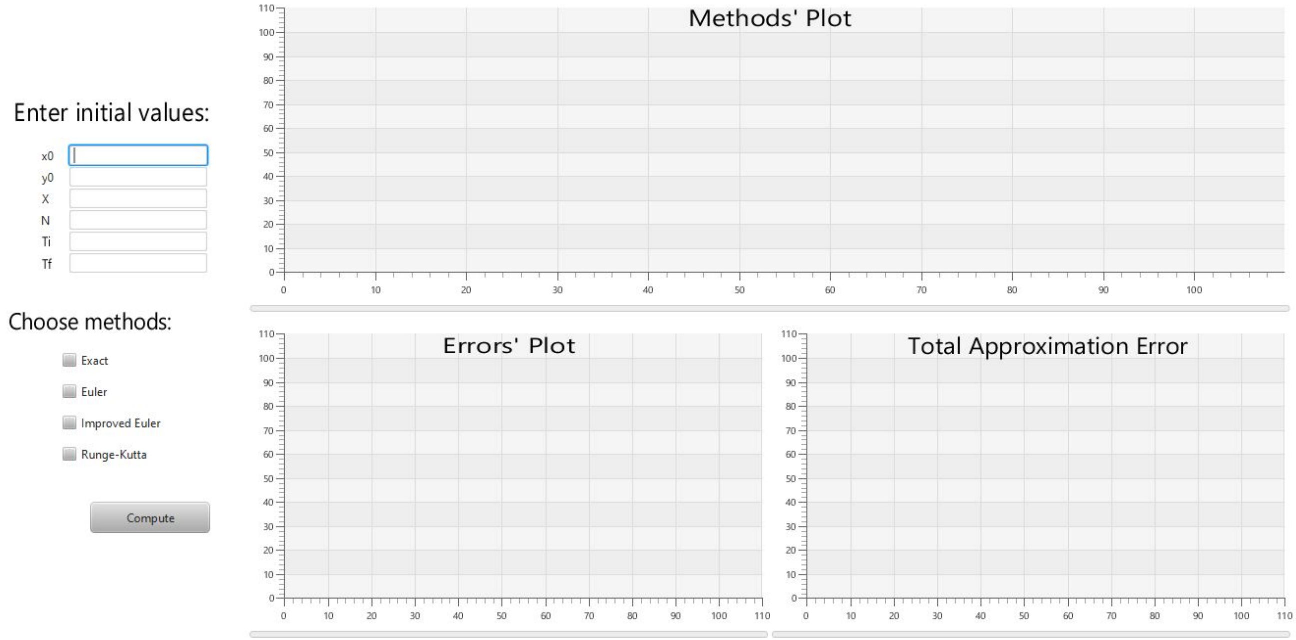
Group 1

Equation №15

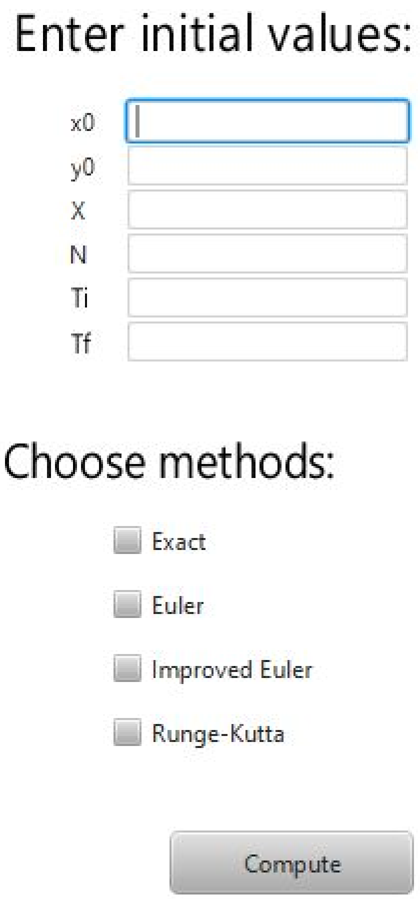
Content:

How to work with GUI

This is how GUI looks like:



Left part:



It gives the possibility to change initial values x_0 and y_0, the maximum value of $x_i - X$, the number of points in the plot $-N$, the initial and final value of $N - T_i$ and T_f to compute total approximation error. Besides this, we can choose methods that we want to see on the plot by putting a checkmark to the corresponding method's box.
To see graphs, you should press button *Compute,* at the top will be a plot of all methods that you chose, bellow plot of errors and plot of total approximation error of corresponding methods.

General information

1) I implemented assignment on Java programming language using JavaFX library to deal with GUI.
2) To run a program, you should run *Main.java,* which will call function *computation()* in class *Controller* after button *Compute* will be pressed

```
@FXML
void computation() {
// Clear charts before start
    methodsChart.getData().clear();
    errorsChart.getData().clear();
    approxChart.getData().clear();
// Take values from a text fields
    double x = Double.parseDouble(x0.getCharacters().toString());
    double y = Double.parseDouble(y0.getCharacters().toString());
    double X = Double.parseDouble(maxX.getCharacters().toString());
    int N = Integer.parseInt(n.getCharacters().toString());
    int initial = Integer.parseInt(Ti.getCharacters().toString());
    int finalN = Integer.parseInt(Tf.getCharacters().toString());
// Plot corresponding graph
    if (isExact.isSelected()) {
    // Create object of class Exact
        Exact ex = new Exact(x, y, X, N);
    // Call function to compute and get Series with solution and add it to the method's chart
        methodsChart.getData().add(ex.solveExact());
    }
```

```java
if (isEuler.isSelected()) {
// Create object of class Euler
    Euler eu = new Euler(x, y, X, N);
// Call function to compute and get Series with solution and add it to the method's chart
    methodsChart.getData().add(eu.solveEuler());
// To calculate errors exact have to been computed
    if (isExact.isSelected()) {
    // Call function to get Series with errors and add it to the error's chart
        errorsChart.getData().add(eu.getErrors());
    // Call function to get Series with total approximation errors and add it to the approximation chart
        approxChart.getData().add(eu.getApprox(initial, finalN));
    }
}
if (isImprovedEuler.isSelected()) {
// Create object of class ImprovedEuler
    ImprovedEuler imp = new ImprovedEuler(x, y, X, N);
// Call function to compute and get Series with solution and add it to the method's chart
    methodsChart.getData().add(imp.solveImprovedEuler());
// To calculate errors exact have to been computed
    if (isExact.isSelected()) {
    // Call function to get Series with errors and add it to the error's chart
        errorsChart.getData().add(imp.get_Errors());
    // Call function to get Series with total approximation errors and add it to the approximation chart
        approxChart.getData().add(imp.getApprox(initial, finalN));
    }
}
```

Function *computation():*

1. clean charts from previous graphs
2. take initial values from text fields
3. check for each method: “Was checkmark put on?”
4. if yes, create an object of the corresponding class and add computed graph to methods’ chart
5. for methods only: check that *Exact* was chosen
6. if yes, compute errors and total approximation error and add them to the charts

3) To compute derivative, there is function *F* in *Main*

```java
// Function which computes given y' with particular x and y
    static double F(double x, double y) { return 2 * Math.exp(x) - y; }
```

4) To create an object, there is a constructor in each method class:

1. computation of step

```java
// Compute step
    h = (X - x0) / N;
```

2. computation of x_i

```java
// Computation of x_i
    x_i = new double[N + 1];
    for (int i = 0; i <= N; ++i)
        x_i[i] = x0 + i * h;
```

3. creation of array for y_i

```java
// Create array for y_i
    y_i = new double[N + 1];
```

4. creation of *Series* for methods' and errors' (not in *Exact*) charts and naming them

```
// Create Series
    eulerSeries = new XYChart.Series <Number, Number>();
    eulerSeries.getData().clear();
    eulerErrors = new XYChart.Series <Number, Number>();
    eulerErrors.getData().clear();
    eulerApprox = new XYChart.Series <Number, Number>();
    eulerApprox.getData().clear();
// Give name to Series
    eulerSeries.setName("Euler");
    eulerErrors.setName("Euler");
    eulerApprox.setName("Euler");
```

5) y_i getter in *Exact* (for further errors computation)

```
public static double get_y_i(int i) { return y_i[i];}
```

6) Computation and returning errors in all methods are similar; therefore, as an example, I will show an implementation of the function in class *Euler.* To compute and get errors, we should use function *getErrors()*:

```
XYChart.Series <Number, Number> getErrors() {
// Compute error for ith point and add it to the series
    for (int i = 0; i < y_i.length; ++i) {
        eulerErrors.getData().add(new XYChart.Data <Number, Number>(x_i[i], Math.abs(Exact.get_y_i(i) - y_i[i])));
    }
// Return series for further comparing
    return eulerErrors;
}
```

Function *getErrors()*:

1. compute error using formula from the textbook

$$e_i = y(x_i) - y_i$$

2. add error at the point x_i to the *Series*
3. return errors

7) Computation and returning total approximation error in all methods are similar; therefore, as an example, I will show an implementation of the function in class *Euler.* To compute and get errors, we should use function *getApprox()*

```
XYChart.Series <Number, Number> getApprox(int initial, int finalN) {
    for (int i = initial; i <= finalN; ++i) {
    // First solve exact
        Exact e = new Exact(x0, y0, X, i);
        e.solveExact();
    // Create an object to compute total approximation error
        Euler temp = new Euler(x0, y0, X, i);
        temp.solveEuler();
    // Find max error
        double max = 0;
        for (int j = 0; j < temp.y_i.length; ++j) {
            if (max <= Math.abs(e.get_y_i(j) - temp.y_i[j]))
                max = Math.abs(e.get_y_i(j) - temp.y_i[j]);
        }
    // Add error to the Series
        eulerApprox.getData().add(new XYChart.Data<Number, Number> (i, max));
    }
// Return series for further comparing
    return eulerApprox;
}
```

Function *getApprox(int initial, int finalN)*:

1. to avoid null pointer exception, firstly, compute $y(x_i)$
2. for each number of points from initial to final:
3. compute y_i using already implemented function *solveEuler*
4. find maximum error
5. add it to the *Series*

8) In all examples of charts below, I will use initial values
$x_0 = 0,\ y_0 = 0,\ X = 7,\ N = 10,\ T_i = 1,\ T_f = 100.$

Solution of Exact

On paper

$y' = 2e^x - y$ – this is the first-order linear non-homogeneous equation.

1) First, solve complementary:

   $y' + y = 0$

   $\frac{dy}{y} = -dx$

   $ln|y| = -x + c$

   $y = e^{-x}c(x)$

2) $y' = c'(x)e^{-x} - e^{-x}c(x)$

   $c'(x)e^{-x} - e^{-x}c(x) + e^{-x}c(x) = 2e^x$

   $c'(x) = 2e^{2x}$

   $c(x) = e^{2x} + c$

$y = ce^{-x} + e^x$

From here, we can compute $c = ye^x - e^{2x}$ with known initial values $x0$ and $y0$.

In application

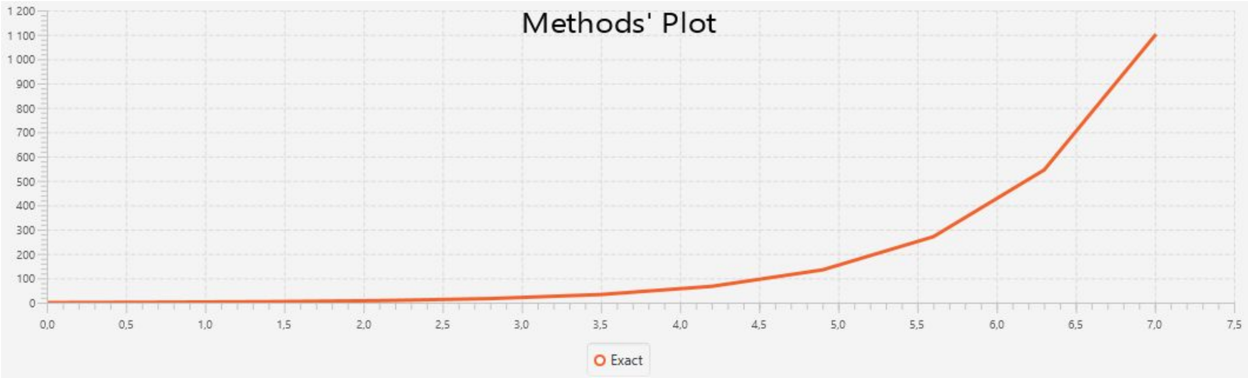
To calculate $y(x_i)$, we should use function *solveExact()* from *class Exact*:

```
XYChart.Series <Number, Number> solveExact() {
// First, compute constant
    double c = y0 * Math.exp(x0) - Math.exp(2 * x0);
// Compute y_i
    for (int i = 0; i < x_i.length; ++i) {
        y_i[i] = Math.exp(x_i[i]) + c * Math.exp(-x_i[i]);
    // Add x_i and y_i to series
        exactSeries.getData().add(new XYChart.Data <Number, Number>(x_i[i], y_i[i]));
    }
// Return series for further comparing
    return exactSeries;
}
```

Function *solveExact()*:

1. compute constant *c*
2. for each x_i compute $y_i = ce^{-x} + e^x$
3. add x_i and y_i to the *Series*
4. return *Series*

Using the result of this function, we have a plot of the solution of Exact method:



Implementation of the Euler method

I have all the implementation of the Euler method in class *Euler*. To calculate $y(x_i)$, we should use function *solveEuler():*

```
XYChart.Series <Number, Number> solveEuler() {
//  Add first point
    eulerSeries.getData().add(new XYChart.Data <Number, Number>(x0, y0));
// Compute y_i
    for (int i = 1; i < x_i.length; ++i) {
        y_i[i] = y_i[i - 1] + h * Main.F(x_i[i - 1], y_i[i - 1]);
    // Add x_i and y_i to series
        eulerSeries.getData().add(new XYChart.Data <Number, Number>(x_i[i], y_i[i]));
    }
// Return series for further comparing
    return eulerSeries;
}
```

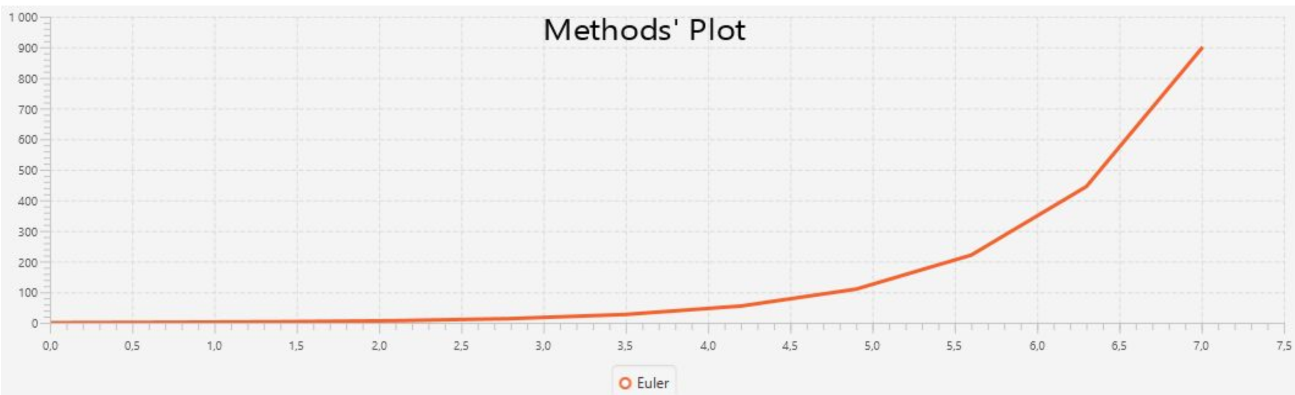
Function *solveEuler():*

1. add point (x_0, y_0)
2. compute y_i for each x_i , using formula from the textbook

$$y_{i+1} = y_i + hf(x_i, y_i), \quad 0 \le i \le n - 1. \tag{3.1.4}$$

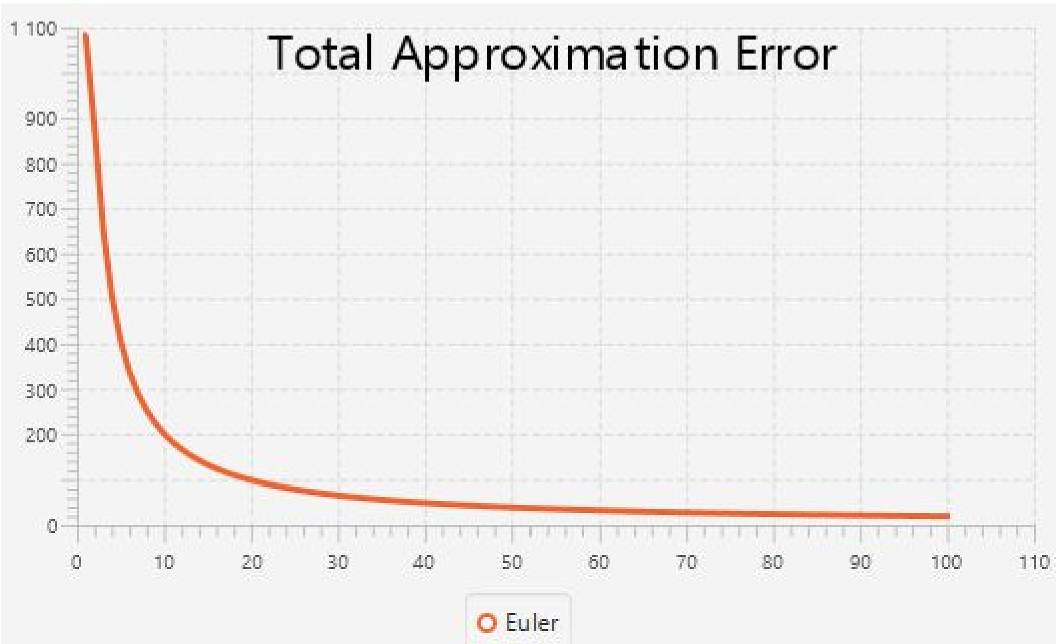
3. add (x_i, y_i) to the *Series*
4. return *Series*

Plot of Euler's method:



Plot of Euler's errors:

Plot of Euler's total approximation error:



Implementation of the Improved Euler method

I have all the implementation of the Improved Euler method in class *ImprovedEuler.* To calculate $y(x_i)$, we should use function *solveImprovedEuler():*

```
XYChart.Series <Number, Number> solveImprovedEuler() {
//  Add first point
    improvedSeries.getData().add(new XYChart.Data<Number, Number>(x0,y0));
// Compute y_i
    for (int i = 1; i < x_i.length; ++i) {
        double k1 = Main.F(x_i[i - 1], y_i[i - 1]);
        double k2 = Main.F(x_i[i],  y: y_i[i - 1] + h * k1);
        y_i[i] = y_i[i - 1] + (h / 2) * (k1 + k2);
    // Add x_i and y_i to series
        improvedSeries.getData().add(new XYChart.Data <Number, Number> (x_i[i], y_i[i]));
    }
// Return series for further comparing
    return improvedSeries;
}
```

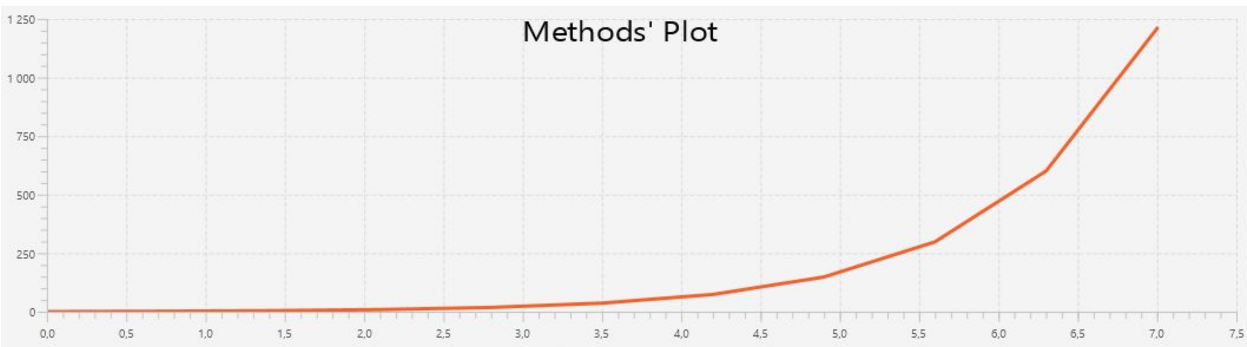
Function *solveImprovedEuler()*:

1. add point (x_0, y_0)
2. compute y_i for each x_i , using formula from the textbook

$$\begin{aligned} k_{1i} &= f(x_i, y_i), \\ k_{2i} &= f\left(x_i + h, y_i + hk_{1i}\right), \\ y_{i+1} &= y_i + \frac{h}{2}(k_{1i} + k_{2i}). \end{aligned}$$

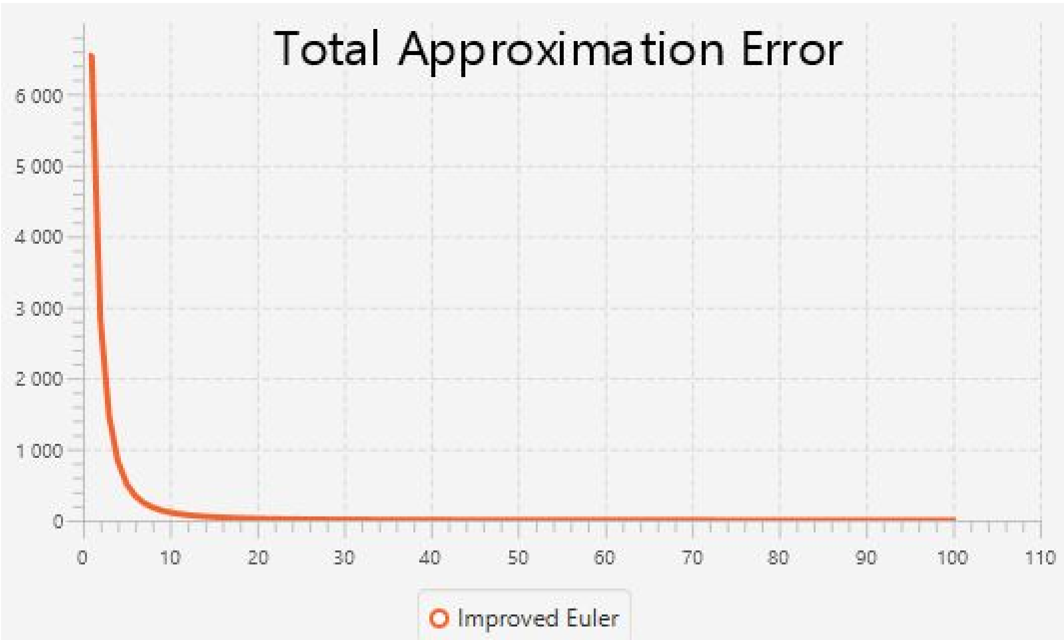
3. add (x_i, y_i) to the *Series*
4. return *Series*

Plot of Improved Euler's method:



Plot of Improved Euler's errors:

Plot of Improved Euler's total approximation error:



Implementation of the Runge-Kutta method

I have all the implementation of the Runge-Kutta method in class *Runge-Kutta.* To calculate $y(x_i)$, we should use function *solveRungeKutta():*

```
XYChart.Series <Number, Number> solveRungeKutta() {
//  Add first point
    RungeKuttaSeries.getData().add(new XYChart.Data <Number, Number>(x0,y0));
// Compute y_i
    for (int i = 1; i < x_i.length; ++i) {
        double k1 = Main.F(x_i[i - 1], y_i[i - 1]);
        double k2 = Main.F( x: x_i[i - 1] + h / 2,  y: y_i[i - 1] + k1 * (h / 2));
        double k3 = Main.F( x: x_i[i - 1] + h / 2,  y: y_i[i - 1] + k2 * (h / 2));
        double k4 = Main.F( x: x_i[i - 1] + h,  y: y_i[i - 1] + k3 * h);
        y_i[i] = y_i[i - 1] + (h / 6) * (k1 + 2 * k2 + 2 * k3 + k4);
    // Add x_i and y_i to series
        RungeKuttaSeries.getData().add(new XYChart.Data<Number, Number>(x_i[i], y_i[i]));
    }
// Return series for further comparing
    return RungeKuttaSeries;
}
```

Function *solveRungeKutta()*:

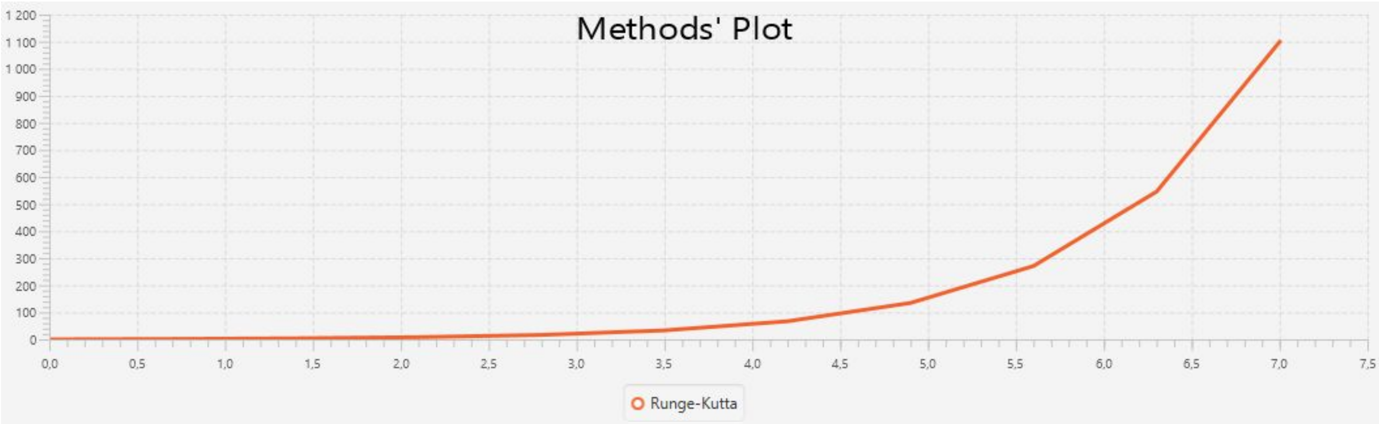
1. add point (x_0, y_0)
2. compute y_i for each x_i , using formula from the textbook

$$\begin{aligned} k_{1i} &= f(x_i, y_i), \\ k_{2i} &= f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_{1i}\right), \\ k_{3i} &= f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_{2i}\right), \\ k_{4i} &= f(x_i + h, y_i + hk_{3i}), \end{aligned}$$

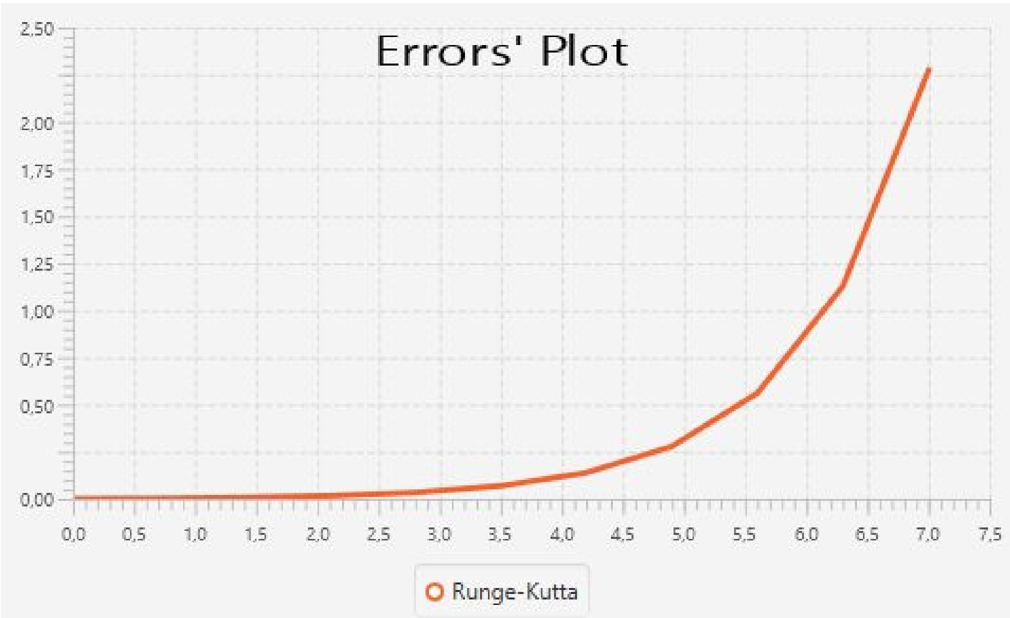
$$y_{i+1} = y_i + \frac{h}{6}(k_{1i} + 2k_{2i} + 2k_{3i} + k_{4i}).$$

3. add (x_i, y_i) to the *Series*
4. return *Series*

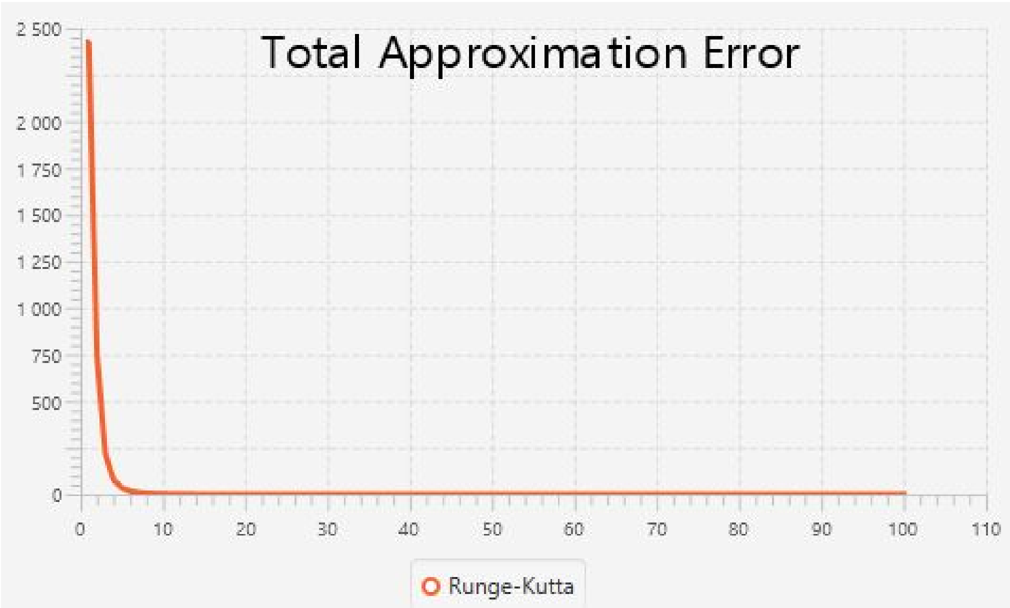
Plot of Runge-Kutta's method



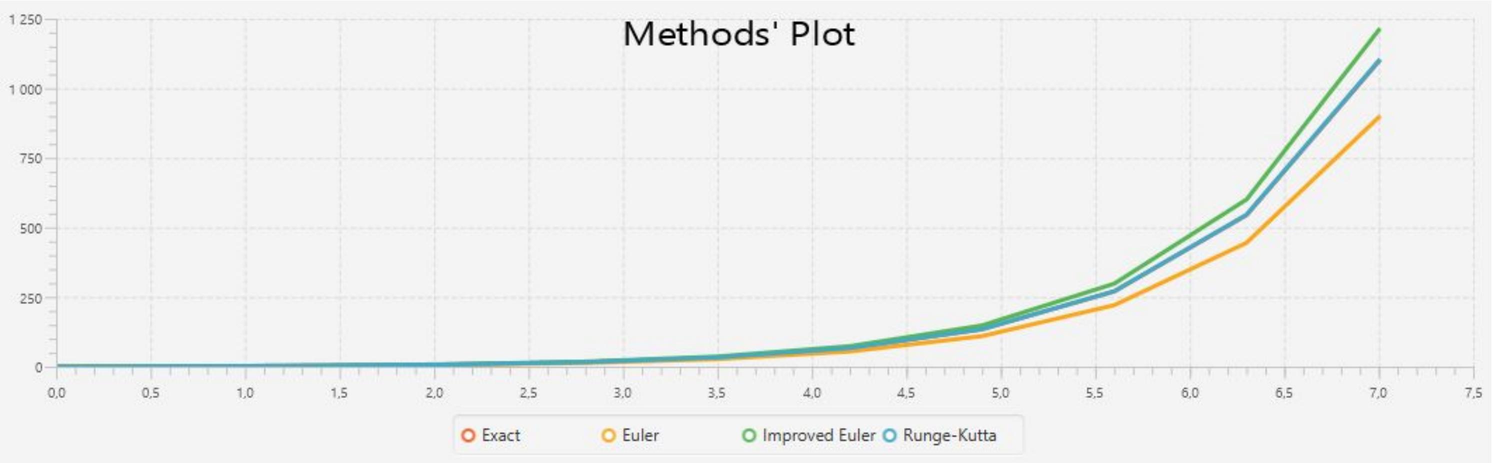
Plot of Runge-Kutta's errors:

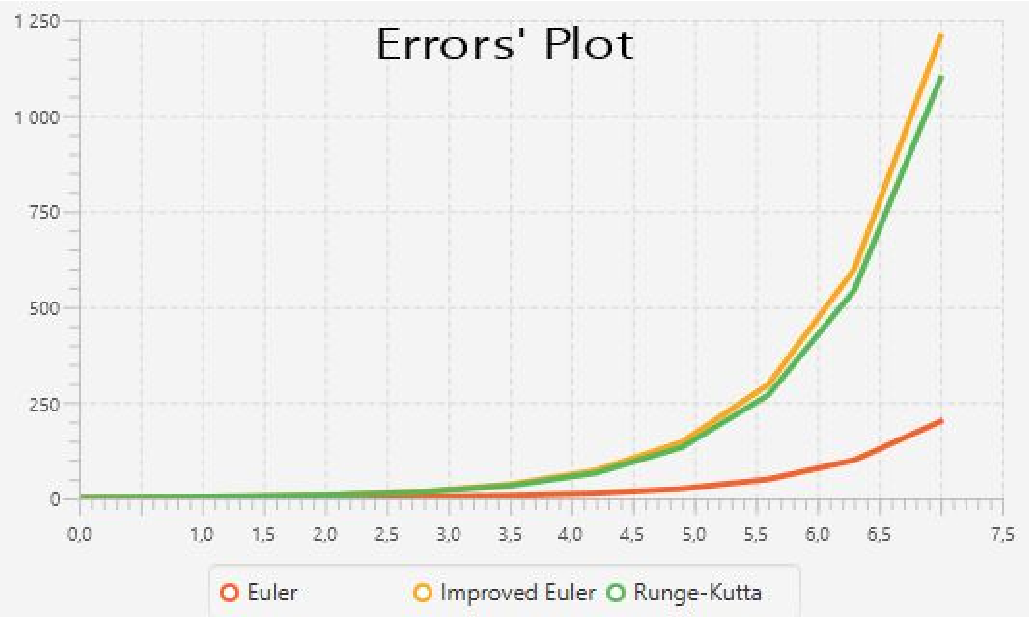


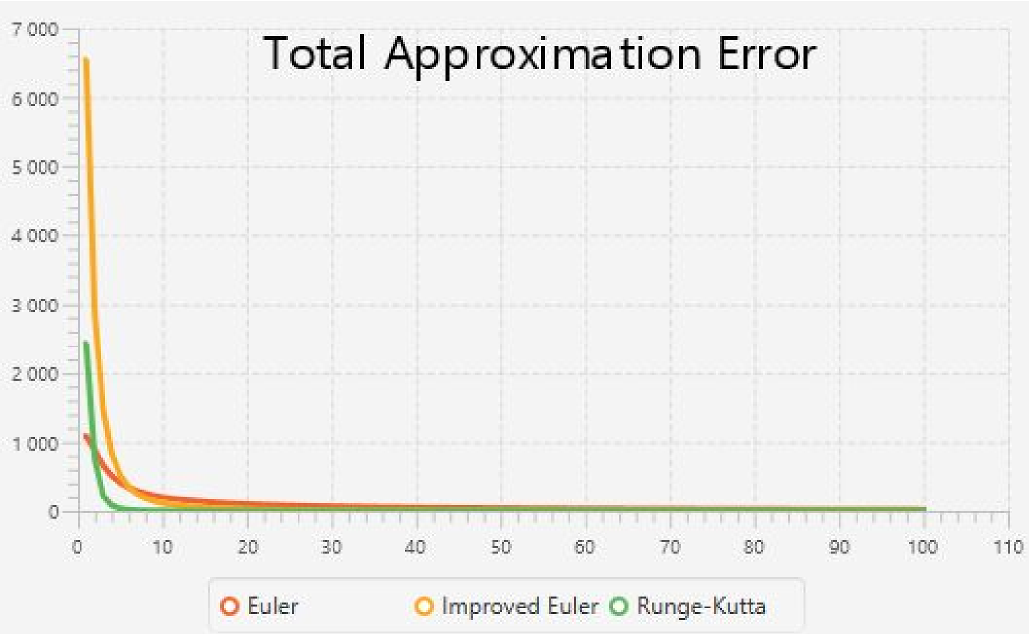
Plot of Runge-Kutta's total approximation error:



Plot of all methods and exact solution:







UML class diagram