Computational Practicum Report

Differential equation

$y' = e^{-sin(x)} - y\ cos(x)$ x0 = 0, y0 = 1

Analytical solution:

1. Classification: first-order linear ordinary differential equation
2. $y' + y\ cos(x) = e^{-sin(x)}$
3. Let $\mu(x) = e^{\int cos(x)dx} = e^{sin(x)}$

   Multiply both sides by $\mu(x)$:

   $e^{sin(x)}y' + y(e^{sin(x)}cos(x)) = 1$
4. $(e^{sin(x)}cos(x)) = \frac{d}{dx}e^{sin(x)} = (e^{sin(x)})'$

   $e^{sin(x)}y' + y(e^{sin(x)})' = 1$
5. $f(x)\ g'(x) + g(x)\ f'(x) = (f(x)g(x)'$

   $(e^{sin(x)}y)' = 1$
6. Integrate both sides:

   $\int(e^{sin(x)}y)'\ dx = \int 1 dx$
7. $e^{sin(x)}y = x + c$
8. Divide both sides by $e^{sin(x)}$:

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

General solution:

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

Solution to IVP:

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

$$y = (x + 1)e^{-sin(x)}$$

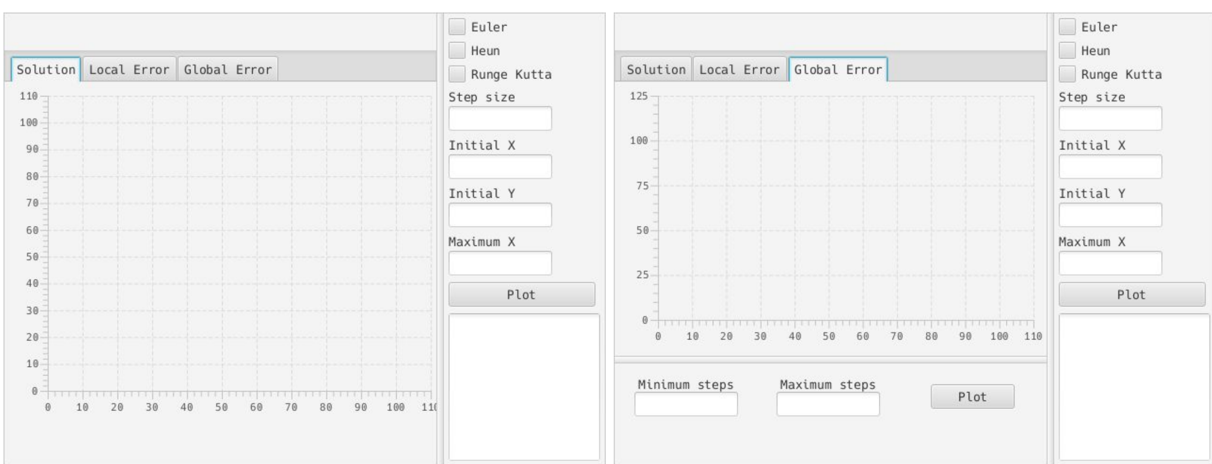
Project structure

Packages:

- App
  - Main.java
  - GUIController.java
  - euler.fxml
- Equation
  - EquationBD.java
  - EquationInterfaceBD.java
- Euler
  - EulerCalculatorBD.java
  - Point.java

Main features of the project:

- Project works mostly in BigDecimal and uses Double only when *Math* functions are needed
  - Reason: Unexpected behavior of Double can affect accuracy in a long run
- Virtually any first-order differential equation (provided it is solved) can be plugged in, as EquationBD implements EquationInterfaceBD



App interface (Local Error tab looks exactly like Solution)

Code overview

EquationBD

- Return the computed value of actual equation for approximation methods

```
/**...*/
public BigDecimal compute(BigDecimal x, BigDecimal y){
    double x1 = x.doubleValue();
    double y1 = x.doubleValue();
    return new BigDecimal( val: Math.exp(-Math.sin(x1)) - y1 * Math.cos(x1), prec);
}
```

- Set constant that will be used to realize the exact solution of DE

```
/**...*/
public void setC(BigDecimal x, BigDecimal y){
    double x1 = x.doubleValue();
    double y1 = y.doubleValue();
    constant = new BigDecimal( val: y1/Math.exp(-Math.sin(x1)) - x1);
}
```

- Return the exact value of equation at a provided point

```
/**...*/
public BigDecimal exact(BigDecimal x) {
    double x1 = x.doubleValue();
    return new BigDecimal( val: (x1 + constant.doubleValue()) * Math.exp(-Math.sin(x1)));
}
```

All methods are established in EquationInterfaceBD and are required to implement any equation we wish to solve. So, to take a look at different equations, only so much lines of code have to be changed.

Point

Simple class for data encapsulation. Has exactly two BigDecimal values, which correspond to x-coordinate and y-coordinate.

EulerCalculatorBD

This is where code gets large enough to not fit on the screen correctly due to peculiar BigDecimal implementation of math operations. For instance, A + B is A.add(B) and so on. All methods of this class return an ArrayList<Point> which can and will be used to plot graphs of solutions. In the core of any method is a loop which runs x-value from initial final X. The difference lies in the way each method computes y-value.

EulerCalculatorBD.exact(equation, finalX)
Used to provide an exact solution to DE. Uses *equation.exact(x)* method to create a point.

EulerCalculatorBD.euler(equation, finalX)
Used to approximate DE solution using Euler method. By Euler method, y-coordinate is computed recursively as follows:
$y_1 = y_0 + step * f(x_0, y_0)$

EulerCalculatorBD.heun(equation, finalX)
Used to approximate DE solution using Heun (otherwise known as Improved Euler) method. By Heun method, y-coordinate is computed with the usage of intermediate-y as follows:
$y_i = y_0 + step * f(x_0, y_0)$
$y_1 = y_0 + (f(x_0, y_0) + f(x_0 + step, y_i)) * step / 2$

EulerCalculatorBD.rungeKutta(equation, finalX)
Used to approximate DE solution using Runge Kutta method. By Runge Kutta method, y-coordinate is computed with the usage of 4 intermediate values as follows:
$k_1 = step * f(x_0, y_0)$
$k_2 = step * f(x_0 + step/2, y_0 + k_1/2)$
$k_3 = step * f(x_0 + step/2, y_0 + k_2/2)$
$k_4 = step * f(x_0 + step, y_0 + k_3)$
$y_1 = y_0 + (k_1 + 2k_2 + 2k_3 + k_4)/6$

GUIController

This class controls the entire app. The two main methods are *execute()* and *calculateGlobalError()* which correspond to exactly two buttons the app has.

execute()

This method is responsible for plotting a chart of solutions and a chart of local errors on the given interval. In order to begin plotting the graph, a method called *check()* runs over all input variables (step size, initial values and final value) and checks them for correctness - there should be only numbers, step should be positive, X and Y are capped at (-100; 100) for the app not to stall. If check is successful, the fun begins. Method *plot()* is invoked, which gathers all input data and builds graphs. Exact solution is always plotted, graphs of approximation methods and corresponding local errors are built only if corresponding CheckBoxes are active.

calculateGlobalError()

It operates mostly the same as *execute()*, only with a slightly different input. This method does not care about the size of step; it is interested in two extra fields on the *Global Error* tab - initial number of steps and maximum number of steps. After running an extra checker (*check2()*) for those two fields, it passes control to *graphGlobalError()*. It gathers all input data and plots a graph of maximum approximation error depending on a number of steps. First, it collects points of all needed methods, then it calculates local error for each, then finds maximum error value and puts it as a point with coordinates (number of steps, error value).

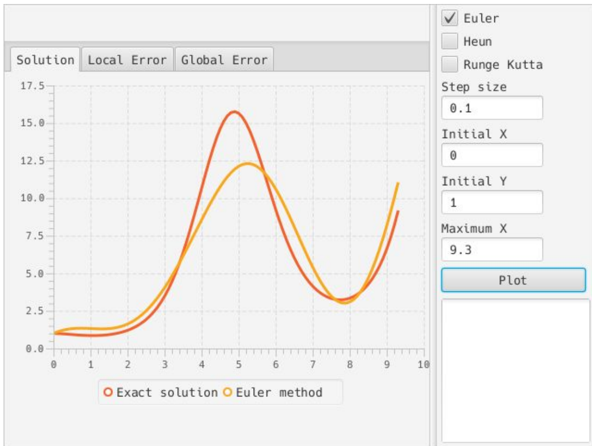
There are also several *private* methods, which, in my opinion, do not really require explanation. Those methods serve the only purpose - to make main methods shorter and more readable.

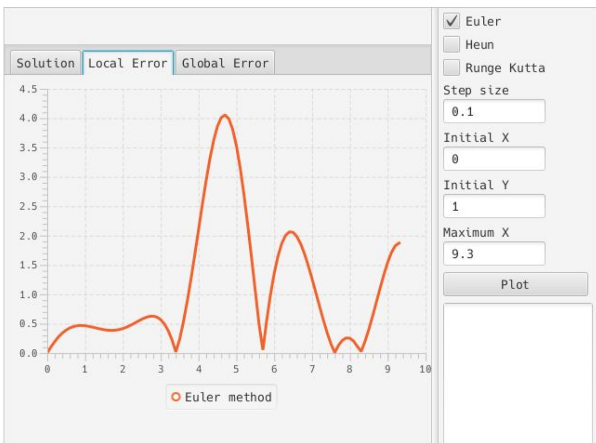
Graphs

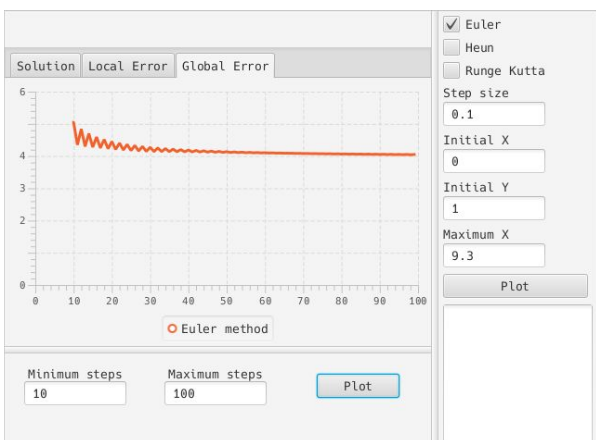
Graphs are built according to given IVP with a step of 0.1:

$y' = e^{-sin(x)} - y\ cos(x)$ x0 = 0, y0 = 1, finalX = 9.3

Euler Method

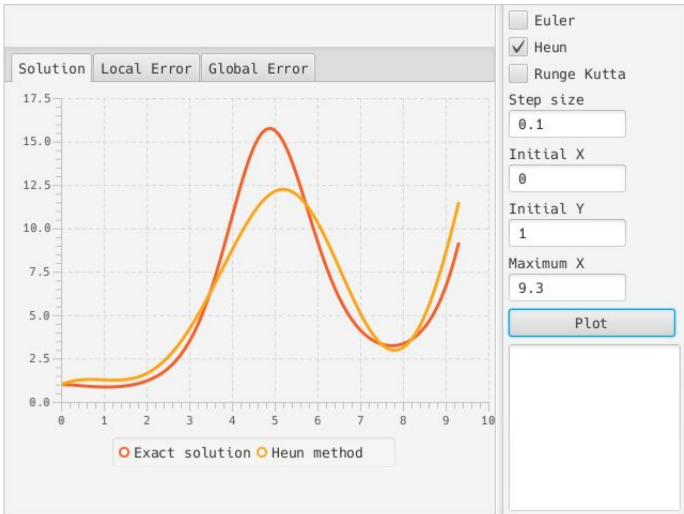


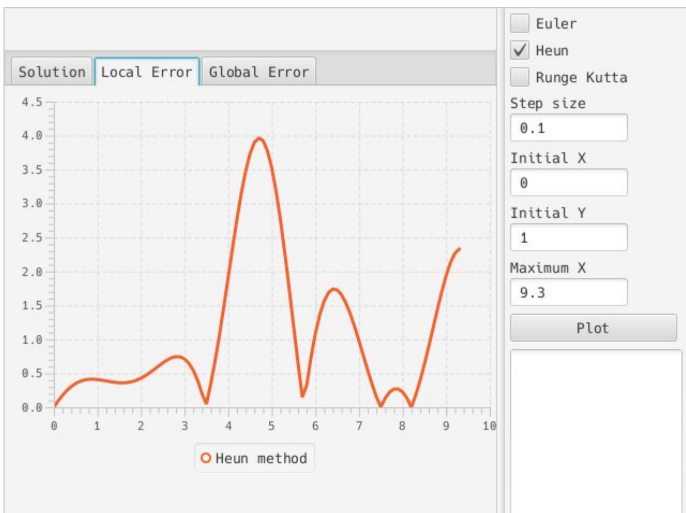


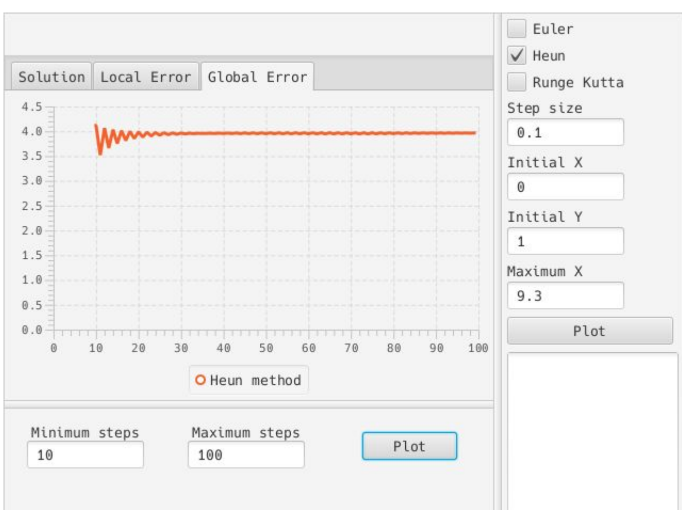


Maximum error converges at 4

Heun Method

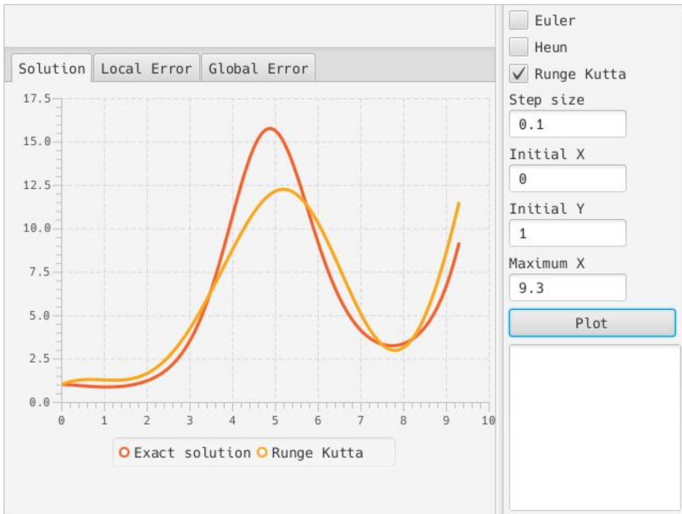


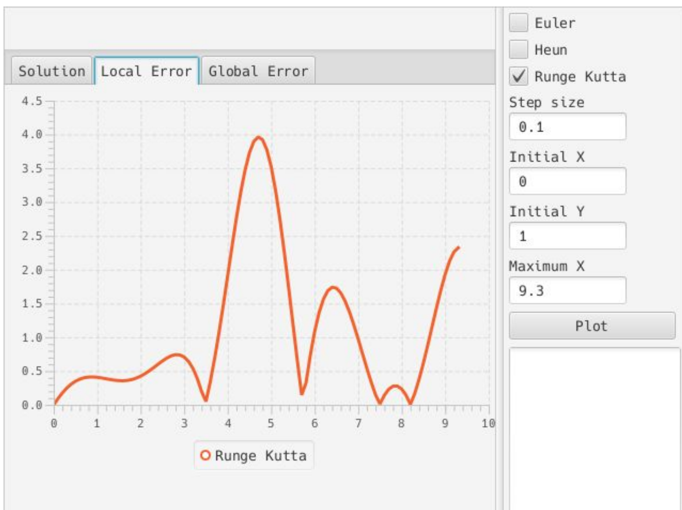


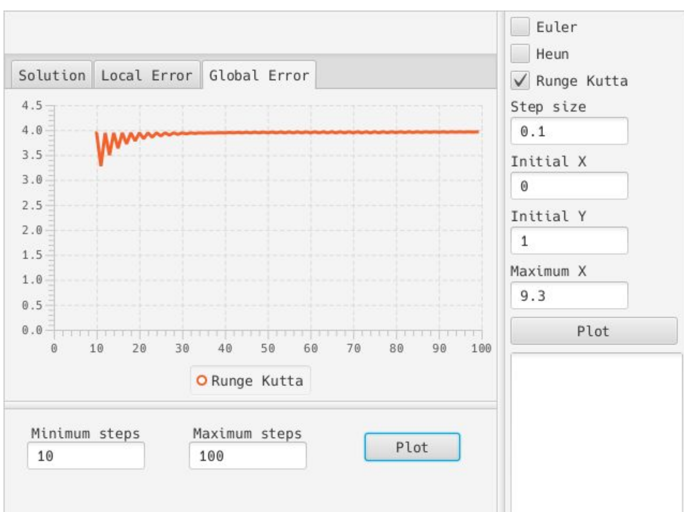


Maximum error converges at 4

Runge Kutta







Maximum error converges at 4

Conclusion

All requirements have been met in this project - the app can provide an exact solution of an IVP with different initial values, graphs can be plotted using different step size and different approximation methods. Also, the methods can be studied in terms of approximation errors using compact graph of global errors which allows to specify different amount of grid steps.