Computational Practicum Report

https://github.com/mintyque/Euler-Calculator

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:

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

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

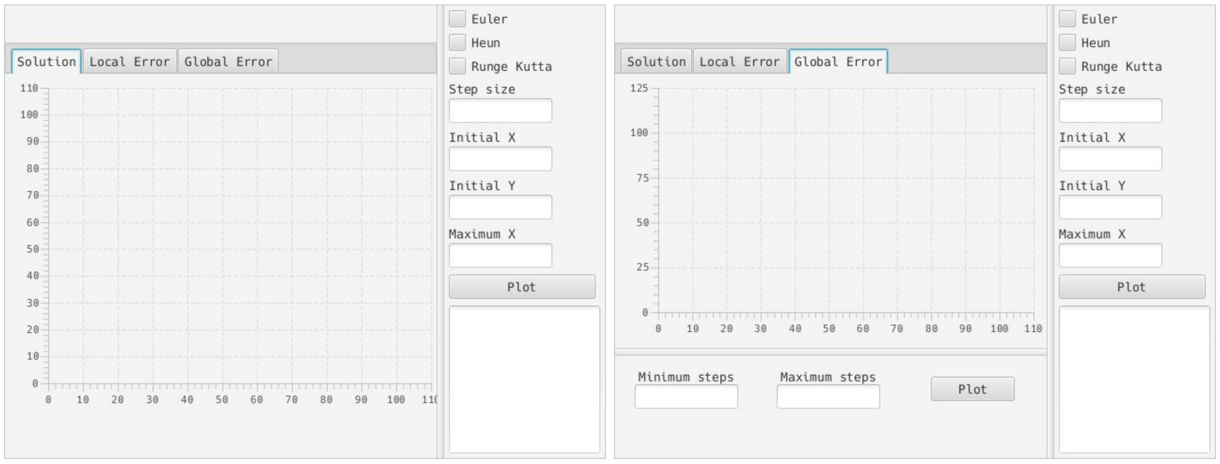
Project structure

Packages:

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

Main features of the project:

- Project uses double as a datatype for holding values and solutions
- Virtually any first-order differential equation (provided it is solved) can be plugged in, as EquationBD implements EquationInterfaceBD
- Neat class structure subject to OOP principles
- ReadMe which specifies some technical aspects of the app

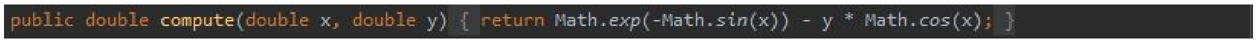


App interface (Local Error tab looks exactly like Solution)

Code overview

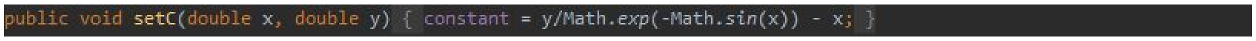
EquationBD

- Return the computed value of actual equation for approximation methods



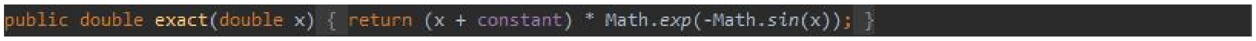
```java
public double compute(double x, double y) { return Math.exp(-Math.sin(x)) - y * Math.cos(x); }
```

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



```java
public void setC(double x, double y) { constant = y/Math.exp(-Math.sin(x)) - x; }
```

- Return the exact value of equation at a provided point



```java
public double exact(double x) { return (x + constant) * Math.exp(-Math.sin(x)); }
```

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.

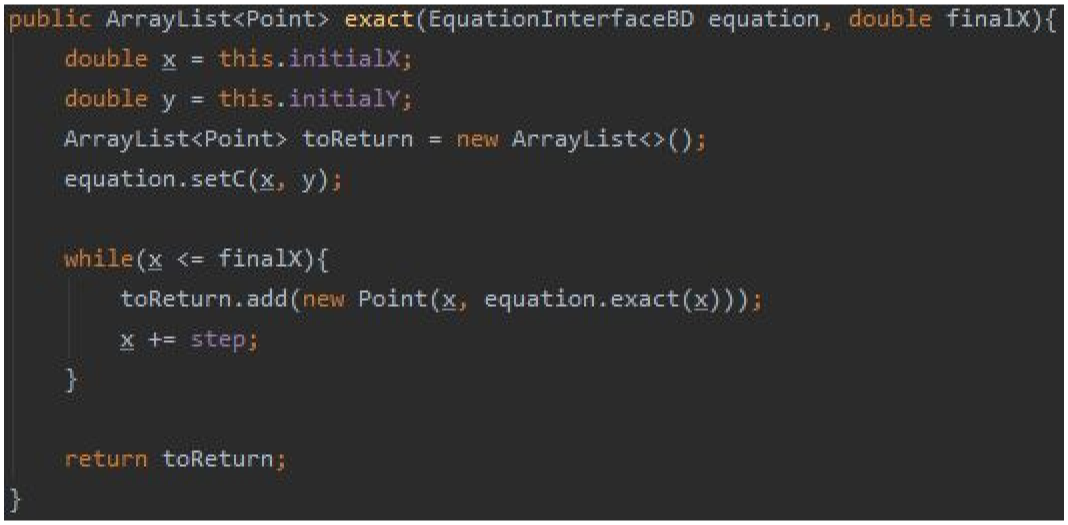
Point

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

EulerCalculatorBD

EulerCalculatorBD.exact(equation, finalX)

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

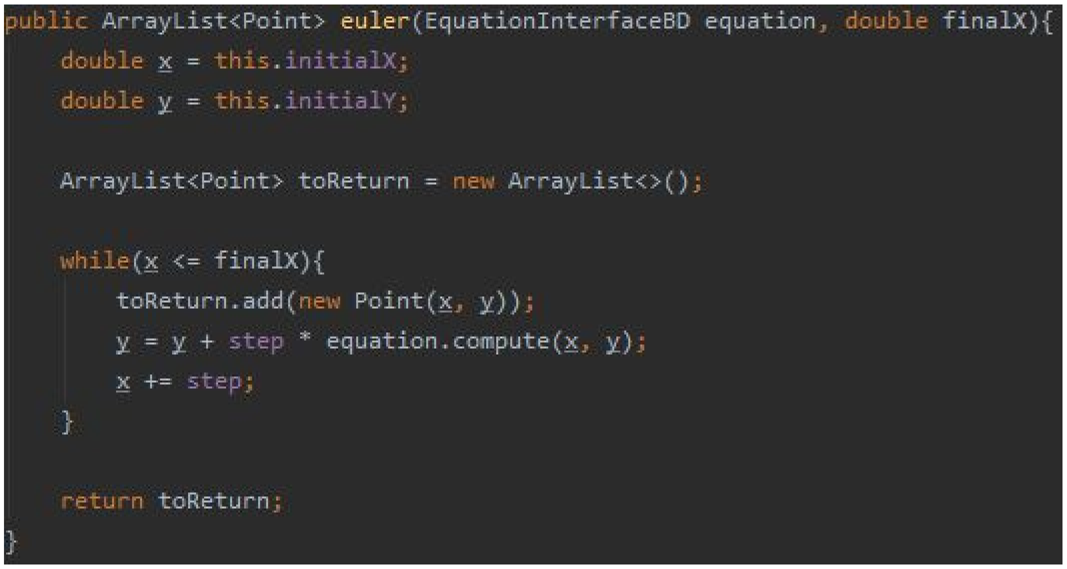


```java
public ArrayList<Point> exact(EquationInterfaceBD equation, double finalX){
    double x = this.initialX;
    double y = this.initialY;
    ArrayList<Point> toReturn = new ArrayList<>();
    equation.setC(x, y);

    while(x <= finalX){
        toReturn.add(new Point(x, equation.exact(x)));
        x += step;
    }

    return toReturn;
}
```

EulerCalculatorBD.euler(equation, finalX)

Used to approximate DE solution using Euler method. By Euler method, y-coordinate is computed recursively as follows:

$y_1 = y_0 + step * f(x_0, y_0)$



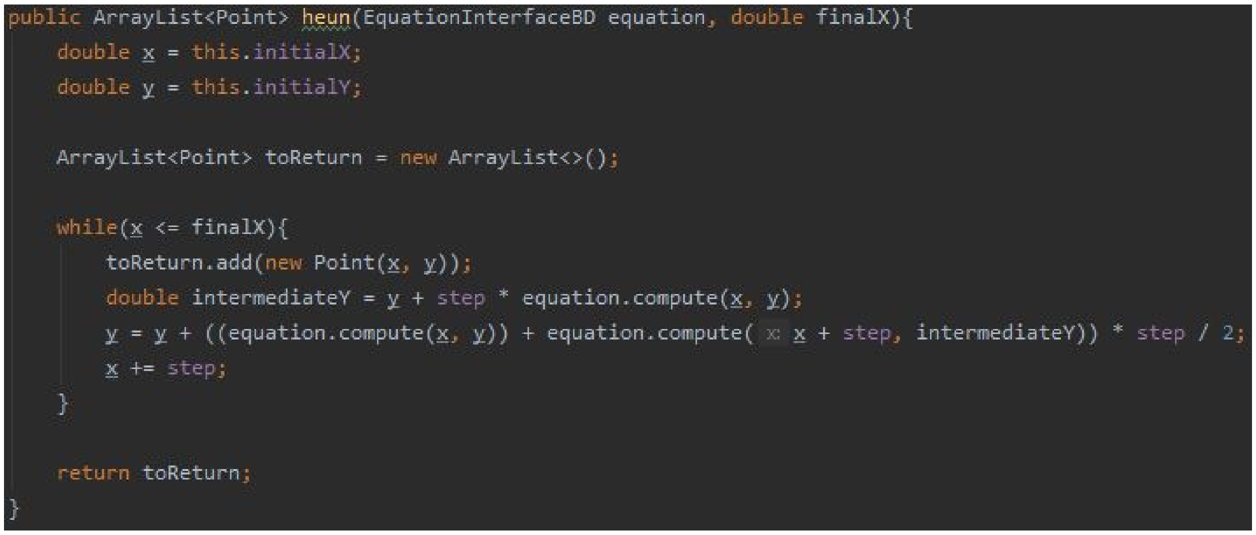
```java
public ArrayList<Point> euler(EquationInterfaceBD equation, double finalX){
    double x = this.initialX;
    double y = this.initialY;

    ArrayList<Point> toReturn = new ArrayList<>();

    while(x <= finalX){
        toReturn.add(new Point(x, y));
        y = y + step * equation.compute(x, y);
        x += step;
    }

    return toReturn;
}
```

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:

$y_i = y_0 + step * f(x_0, y_0)$

$y_1 = y_0 + (f(x_0, y_0) + f(x_0 + step, y_i)) * step / 2$



```java
public ArrayList<Point> heun(EquationInterfaceBD equation, double finalX){
    double x = this.initialX;
    double y = this.initialY;

    ArrayList<Point> toReturn = new ArrayList<>();

    while(x <= finalX){
        toReturn.add(new Point(x, y));
        double intermediateY = y + step * equation.compute(x, y);
        y = y + ((equation.compute(x, y)) + equation.compute( x: x + step, intermediateY)) * step / 2;
        x += step;
    }

    return toReturn;
}
```

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:

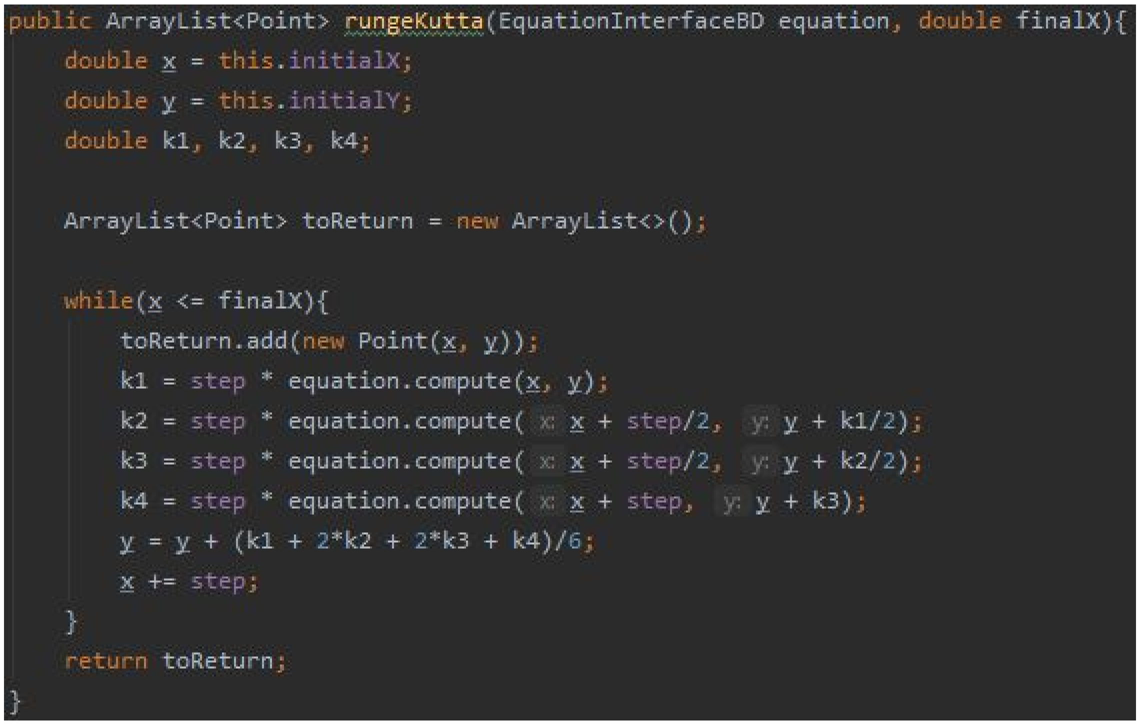
k_1 = step * f(x_0, y_0)

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)

$y_1 = y_0 + (k_1 + 2k_2 + 2k_3 + k_4)/6$



```java
public ArrayList<Point> rungeKutta(EquationInterfaceBD equation, double finalX){
    double x = this.initialX;
    double y = this.initialY;
    double k1, k2, k3, k4;

    ArrayList<Point> toReturn = new ArrayList<>();

    while(x <= finalX){
        toReturn.add(new Point(x, y));
        k1 = step * equation.compute(x, y);
        k2 = step * equation.compute( x: x + step/2,  y: y + k1/2);
        k3 = step * equation.compute( x: x + step/2,  y: y + k2/2);
        k4 = step * equation.compute( x: x + step,  y: y + k3);
        y = y + (k1 + 2*k2 + 2*k3 + k4)/6;
        x += step;
    }
    return toReturn;
}
```

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).

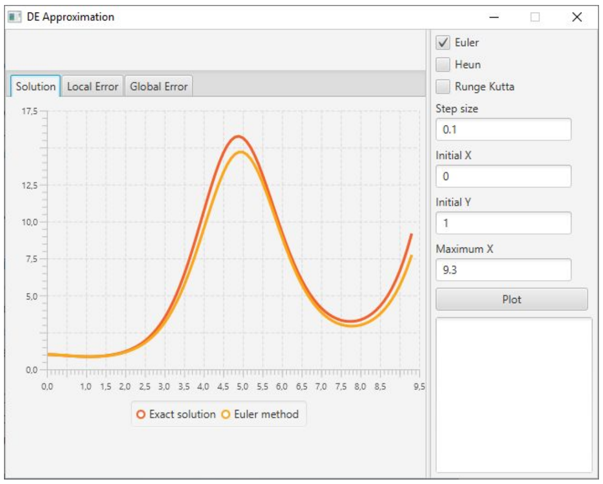
There are also several *private* methods, which, in my opinion, do not really require explanation. Those methods serve the only purpose - to make main methods shorter and more readable.

Graphs

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

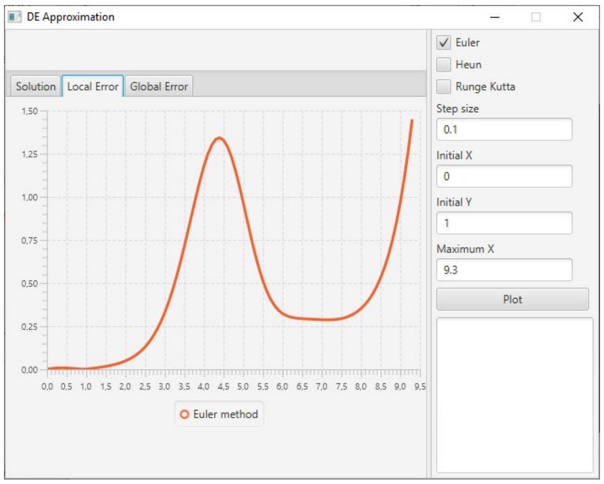
$y' = e^{-sin(x)} - y\, cos(x)$ x0 = 0, y0 = 1, finalX = 9.3

Euler Method



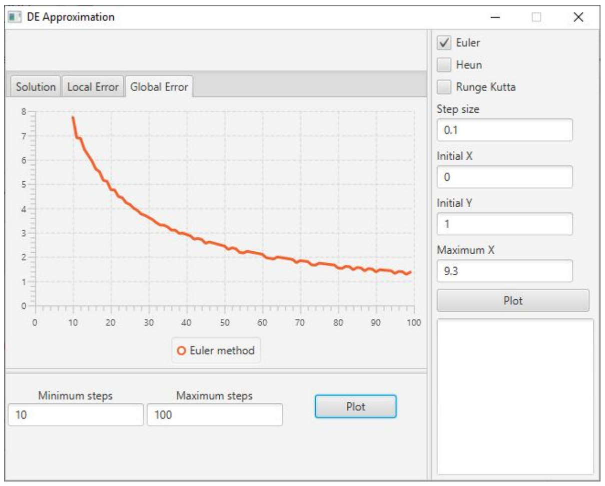
[✓ Euler] Heun Runge Kutta

Step size: 0.1
Initial X: 0
Initial Y: 1
Maximum X: 9.3



[✓ Euler] Heun Runge Kutta

Step size: 0.1
Initial X: 0
Initial Y: 1
Maximum X: 9.3



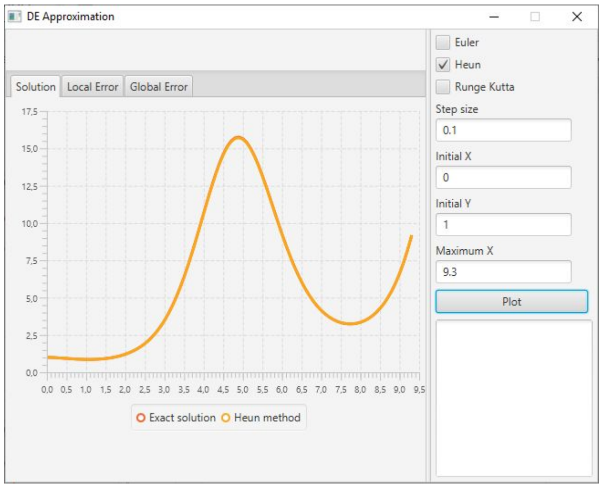
[✓ Euler] Heun Runge Kutta

Step size: 0.1
Initial X: 0
Initial Y: 1
Maximum X: 9.3

Minimum steps: 10
Maximum steps: 100

Maximum error attempts to come closer to 0, but fails due to a small amount of steps. If one would try to decrease a step even more, it would be possible to lower the error, but the app would work for a longer time, so the method is not very optimal for precise calculations

Heun Method



Euler

✓ Heun

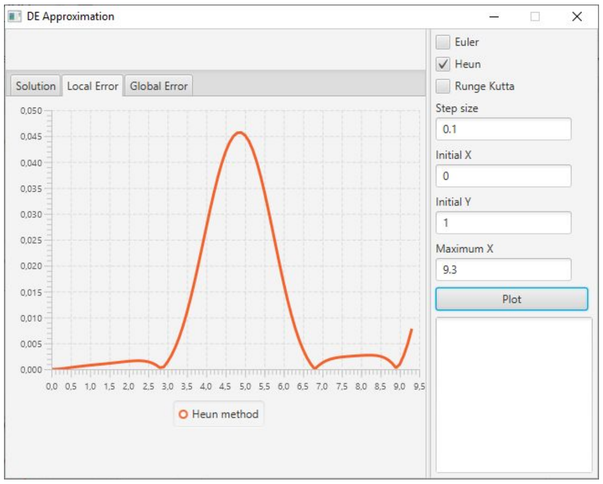
Runge Kutta

Step size: 0.1

Initial X: 0

Initial Y: 1

Maximum X: 9.3



Euler

✓ Heun

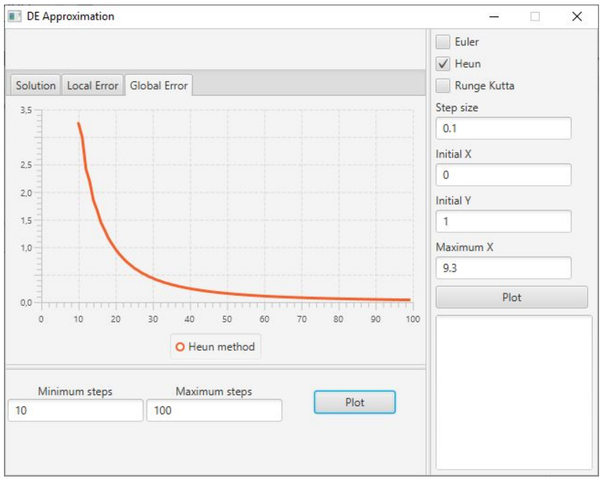
Runge Kutta

Step size: 0.1

Initial X: 0

Initial Y: 1

Maximum X: 9.3



Euler

✓ Heun

Runge Kutta

Step size: 0.1

Initial X: 0

Initial Y: 1

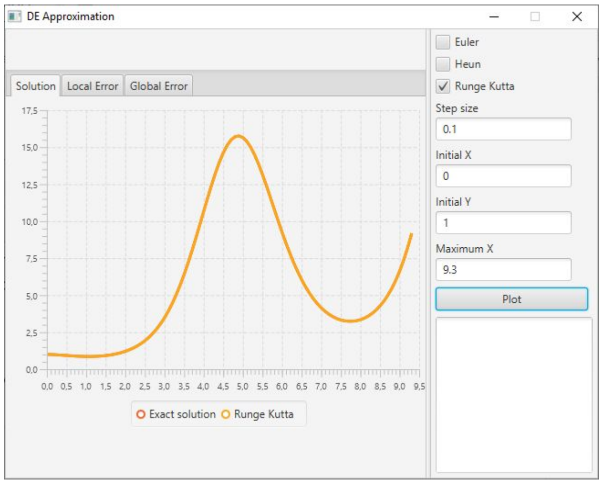
Maximum X: 9.3

Minimum steps: 10

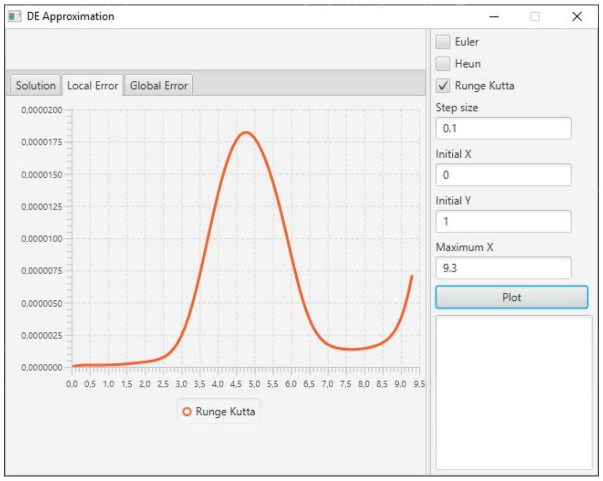
Maximum steps: 100

Maximum error converges at 0. The spike in accuracy is visible, but the error doesn't go over 0.05

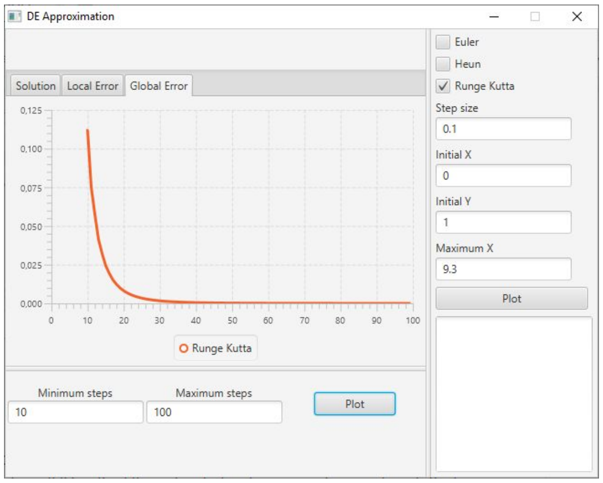
Runge Kutta



☐ Euler
☐ Heun
✓ Runge Kutta
Step size: 0.1
Initial X: 0
Initial Y: 1
Maximum X: 9.3



☐ Euler
☐ Heun
✓ Runge Kutta
Step size: 0.1
Initial X: 0
Initial Y: 1
Maximum X: 9.3



Minimum steps: 10
Maximum steps: 100

☐ Euler
☐ Heun
✓ Runge Kutta
Step size: 0.1
Initial X: 0
Initial Y: 1
Maximum X: 9.3

Maximum error converges at 0. Again, even though there is a spike in accuracy, the error is less than 0.00002, which is the best among methods.

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.