Numerical methods for differential equations solving

Denis Chernikov – B16-04, Differential Equations [S18], Innopolis University

Variant 4

Overview

This work was done in order to understand the principles of numerical methods of differential equations solving, get familiar with their pros and cons and analyze whether it is better to use one or the other.

There were considered three methods for solving Initial Value Problem (Cauchy problem) of the differential equation of the form $y' = f(x, y)$:

- Euler's method
- Improved Euler's method
- Runge-Kutta method

For the purposes of finding out the effectiveness of this methods we need to compare them with the original solution. Next you will see an analytical solution of the given equation.

Analytical exact solution

Initially we're given the equation $y' = 1 + y\frac{2x-1}{x^2}$. Initial Value Problem parameters are $x_0 = 1$ and $y_0 = 1$ ($y(x_0) = y_0$). (By default solution should be considered with $x \in [x_0; X], X = 10$.) Let's find $y(x)$ from this equation.

$$y' - y\frac{2x-1}{x^2} = 1$$

Important note – $x \neq 0$ (function's breakpoint). It is linear non-homogeneous o. d. e. of the form $y'h(x) + yg(x) = f(x)$. As we know, it may be solved by finding the solution for the complement one (with $f(x) = 0$) and finding coefficient (as a function of x) by substituting the answer into the initial equation. Here is the solution of the complement:

$$y' - y\frac{2x-1}{x^2} = 0 \Leftrightarrow$$

$$\Leftrightarrow \frac{dy}{dx} = y\frac{2x-1}{x^2} \Leftrightarrow \cdots$$

considering $y = 0$ is not a solution of the initial one

$$\ldots \Leftrightarrow \frac{dy}{y} = \left(\frac{2x}{x^2} - \frac{1}{x^2}\right)dx \Leftrightarrow$$

$$\Leftrightarrow \int \frac{1}{y}dy = \int \left(\frac{2}{x} - x^{-2}\right)dx \Leftrightarrow$$

$$\Leftrightarrow \ln|y| = 2\ln|x| - (-x^{-1}) + C(x) \Leftrightarrow$$

$$\Leftrightarrow e^{\ln|y|} = e^{\ln|x^2| + 1/x + C(x)} \Leftrightarrow$$

$$\Leftrightarrow y = x^2 e^{1/x} C(x)$$

At this step we need to find $C(x)$ by substituting it into the initial equation. But first of all, let's find $y'(x)$:

$$y' = \left(\left(x^2 e^{1/x}\right)C(x)\right)' = \left(x^2 e^{x^{-1}}\right)'C(x) + x^2 e^{1/x}C'(x) =$$

$$= 2xe^{1/x}C(x) + (x^{-1})'x^2e^{1/x}C(x) + x^2e^{1/x}C'(x) =$$

$$= C'(x)x^2e^{1/x} + C(x)2xe^{1/x} - C(x)\frac{1}{x^2}x^2e^{1/x} =$$

$$= C'(x)x^2e^{1/x} + C(x)2xe^{1/x} - C(x)e^{1/x}$$

Here comes the substitution:

$$C'(x)x^2e^{1/x} + C(x)2xe^{1/x} - C(x)e^{1/x} - x^2e^{1/x}C(x)\frac{2x-1}{x^2} = 1 \Leftrightarrow$$

$$\Leftrightarrow C'(x)x^2e^{1/x} + C(x)e^{1/x}\left(2x - 1 - x^2\frac{2x-1}{x^2}\right) = 1 \Leftrightarrow$$

$$\Leftrightarrow C'(x)x^2e^{1/x} + C(x)e^{1/x}(2x - 1 - 2x + 1) = 1 \Leftrightarrow$$

$$\Leftrightarrow C'(x)x^2e^{1/x} = 1 \Leftrightarrow C'(x) = \frac{1}{x^2e^{1/x}} \Rightarrow$$

$$\Rightarrow C(x) = \int x^{-2}e^{-x^{-1}} = e^{-x^{-1}} + C_1 = \frac{1}{e^{1/x}} + C_1$$

Finally, let's substitute $C(x)$ into the found $y(x)$:

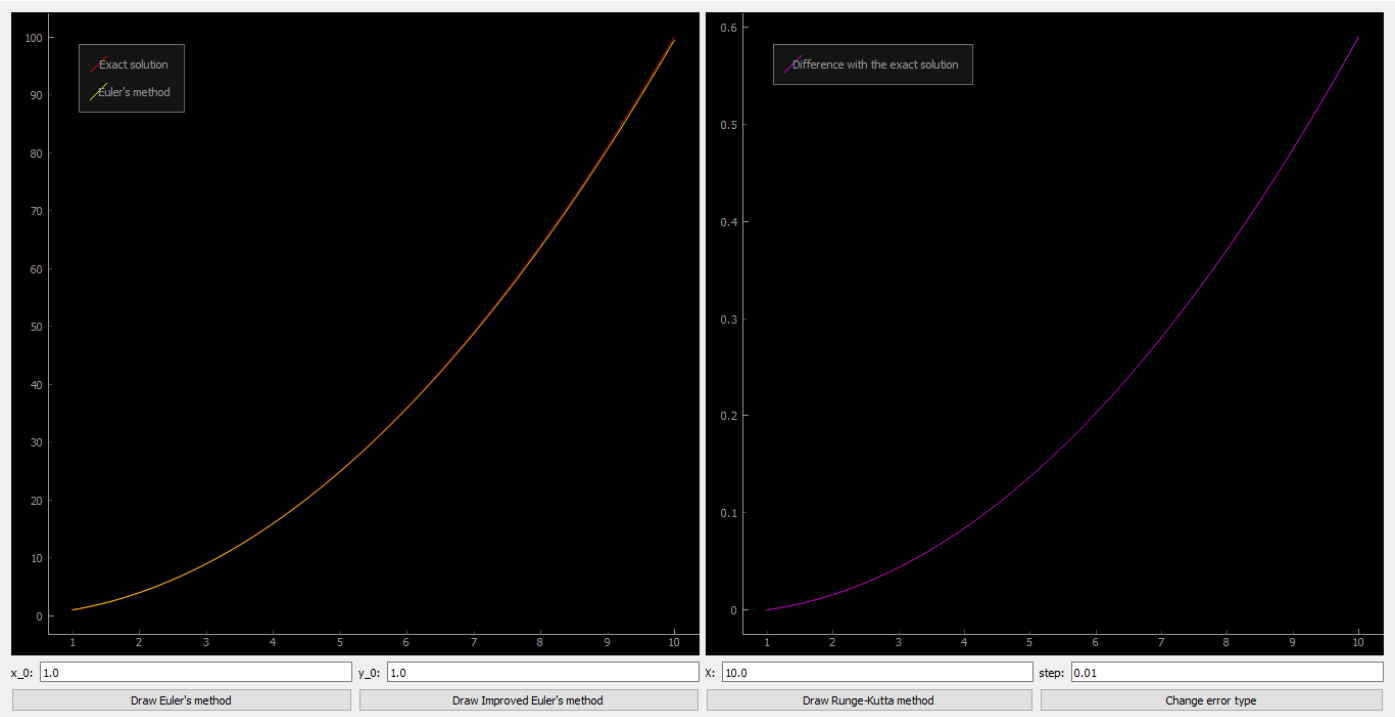
$$y = x^2e^{1/x}C(x) = x^2e^{1/x}\left(\frac{1}{e^{1/x}} + C_1\right) = x^2\left(1 + C_1e^{1/x}\right)$$

Therefore, the solution is $y = x^2\left(1 + C_1e^{1/x}\right), x \neq 0$. For the purposes of solving the IVP we need to find C_1 from the answer: $C_1 = \frac{y_0 - x_0^2}{x_0^2e^{1/x_0}}, x \neq 0$. Given IVP as $y(1) = 1$ we find that $C_1 = \frac{1-1^2}{1^2e^{1/1}} = \frac{0}{e} = 0$ and $y_{\text{IVP}} = x^2, x \neq 0$.

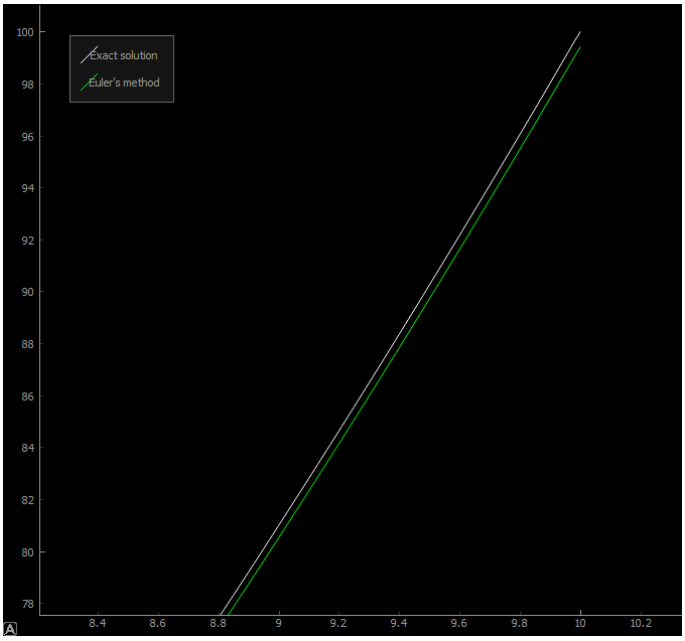
Methods

Euler's method

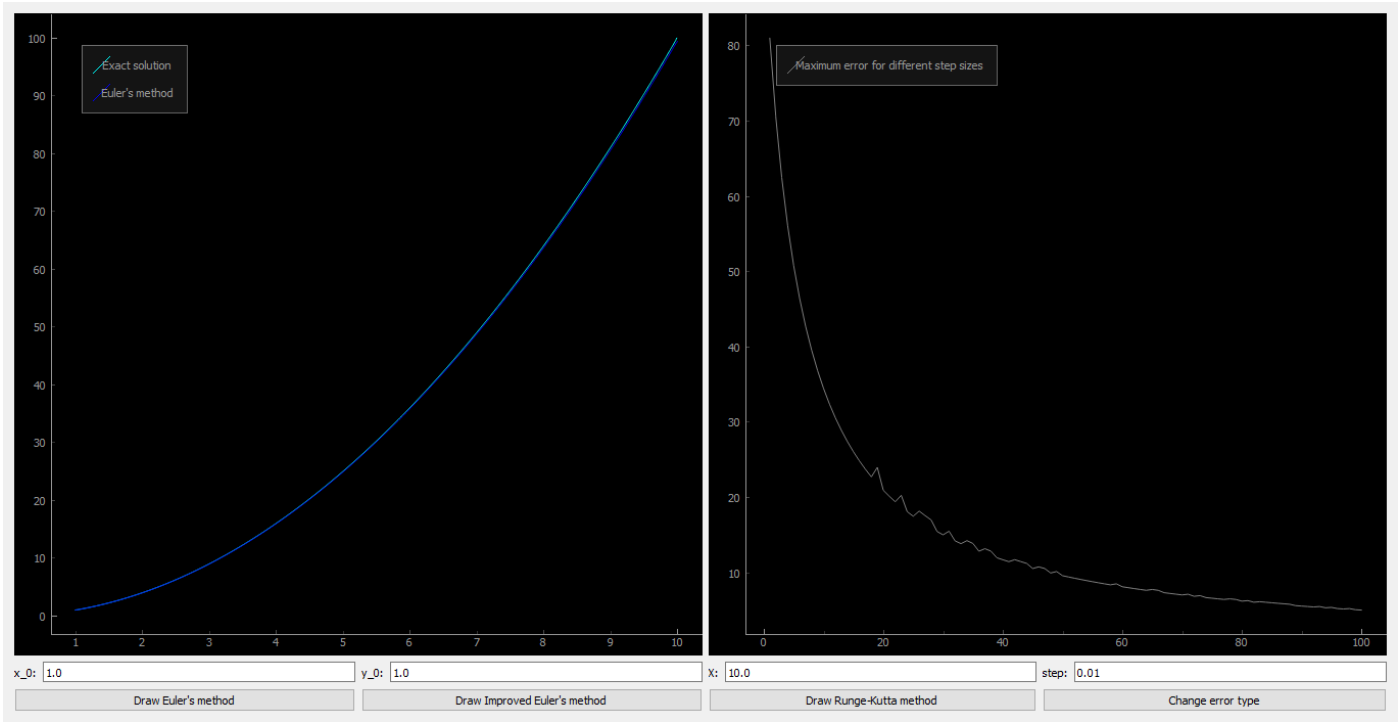
It uses approximation using step, last point and current value of $f(x, y)$. Maximum error with default values is about 0.6. Here you can see plots (for both plots horizontal one is x value; vertical one – for left plot is y value, for right plot is $error$ value):



Error increases with step number (or x value) of a function. Here you can see the difference better:



Next on the right (after pressing the button of changing the error plot type) you can see the dependence of maximum error (vertical axis) on a number of steps used to calculate functions (horizontal axis):



As number of steps increases, maximum error value also goes down.

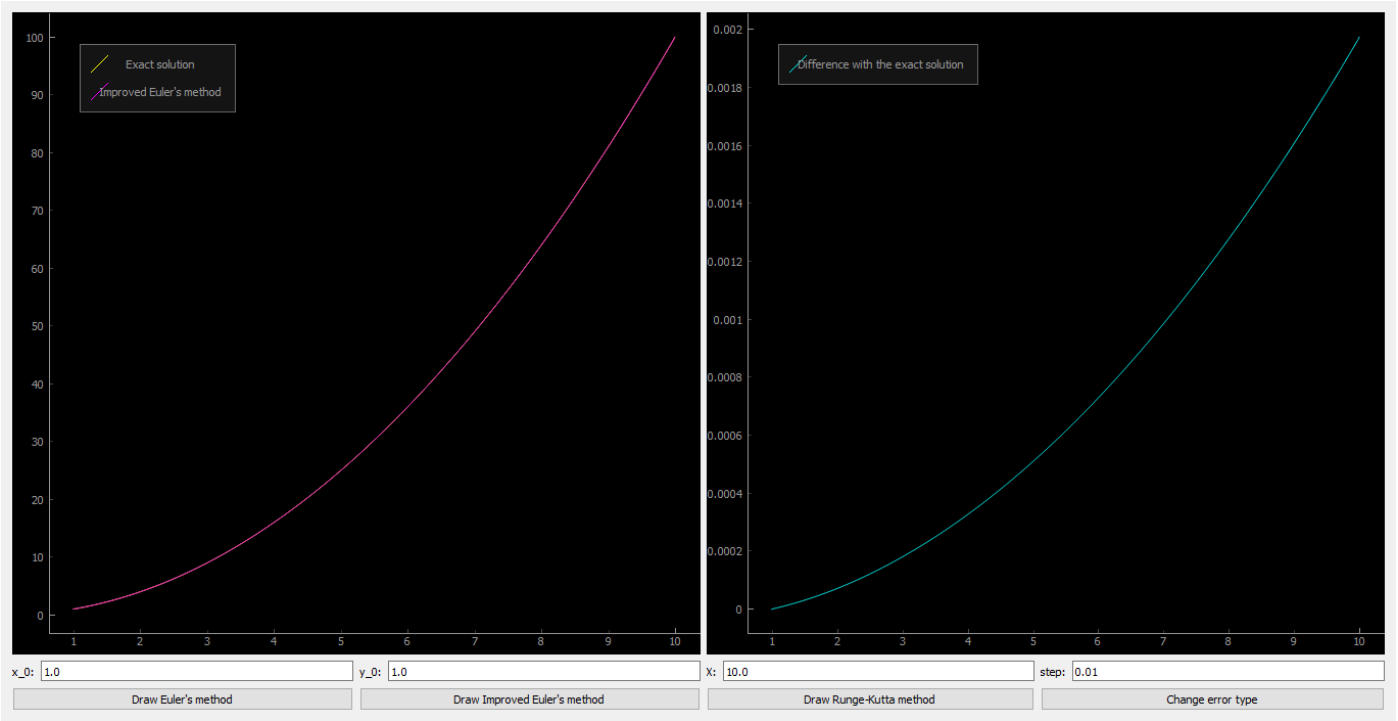
Here is part of code:

```
def euler(x_0, y_0, start, end, step):
    xs = [x_0]
    ys = [y_0]
    if start != x_0:
        xs = [start]
        ys = [y(start, c(x_0, y_0))]
    for i, x in enumerate(rational_range(start + step, end + step, step)):
        xs.append(x)
        ys.append(ys[i] + (x - xs[i]) * f(xs[i], ys[i]))
    return xs, ys
```

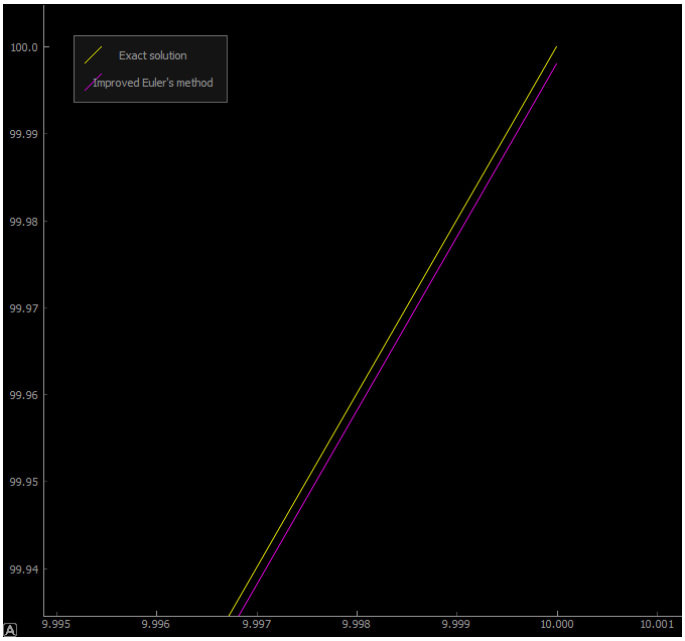
Important to note that for calculations around breakpoint we cannot continue calculations according to the previous values of a function, and we also need to know the value to start from. That's why if start of calculation is not x_0 point we have to calculate y value using exact solution. The other way was to restrict the breakpoint inside the given range, but it is less interesting to complete. For default case everything calculates fairly.

Improved Euler's method

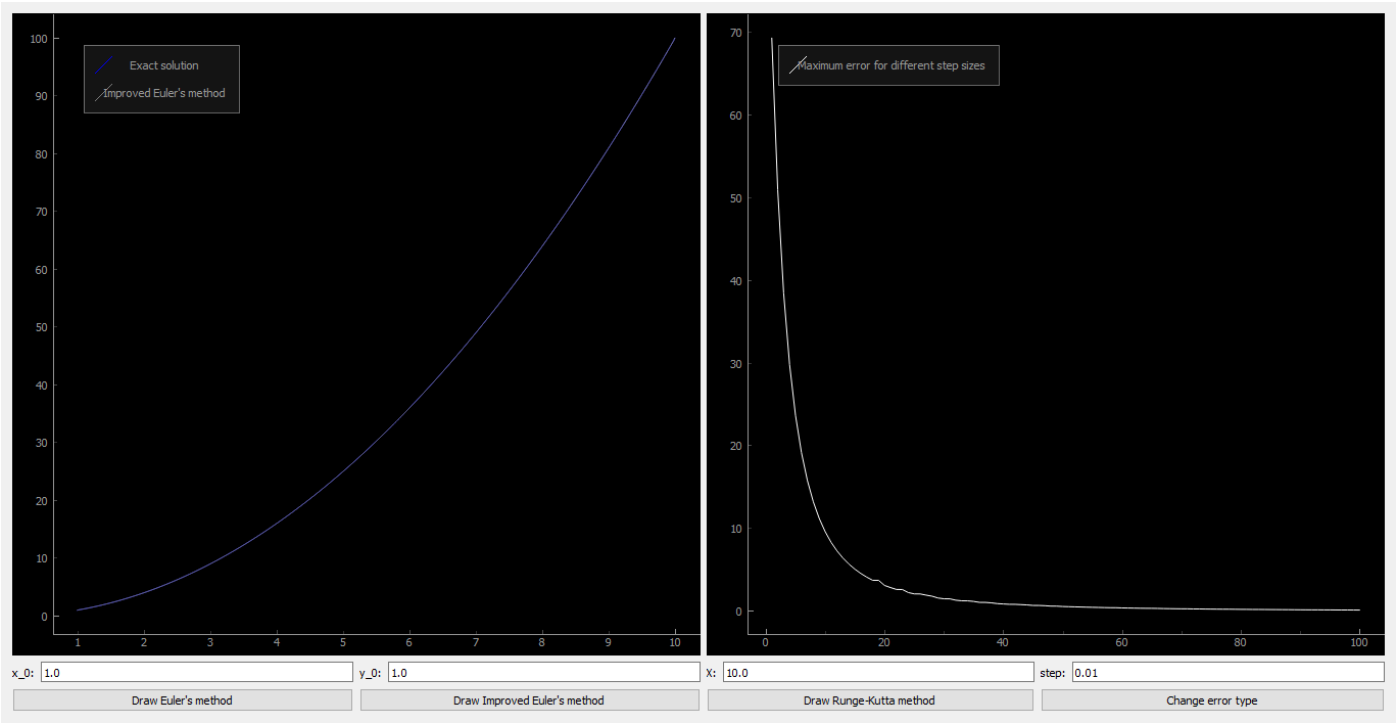
This method improves approximation taking medium between current and previous y values. Here are function plots and error plot (axes are like for previous method):



Maximum error with default inputs is about 0.002. Closer look to see the difference (it is visible only with zoom of 200 times and more):



And here is the dependence of maximum error on number of steps:



Maximum error value is decreasing while the number of steps raises (like for the previous method).

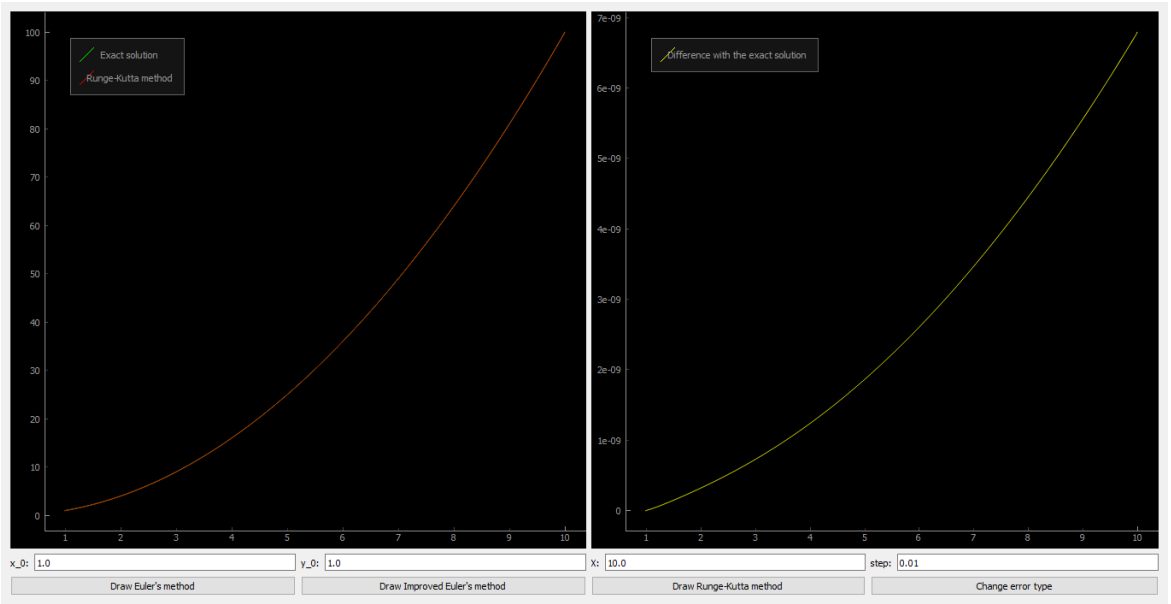
Here is code for that method:

```
def euler_improved(x_0, y_0, start, end, step):
    xs = [x_0]
    ys = [y_0]
    if start != x_0:
        xs = [start]
        ys = [y(start, c(x_0, y_0))]
    for i, x in enumerate(rational_range(start + step, end + step, step)):
        xs.append(x)
        y_pred = ys[i] + (x - xs[i]) * f(xs[i], ys[i])
        ys.append(ys[i] + (x - xs[i]) * (f(xs[i], ys[i]) + f(x, y_pred)) / 2)
    return xs, ys
```

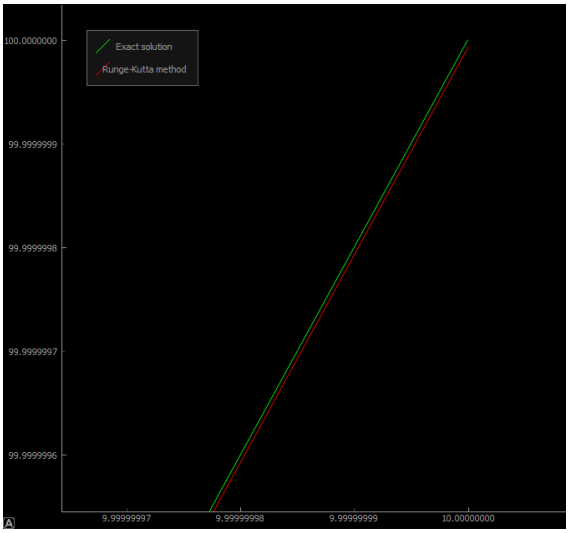
Situation with calculation around the breakpoint is like in previous method.

Runge-Kutta method

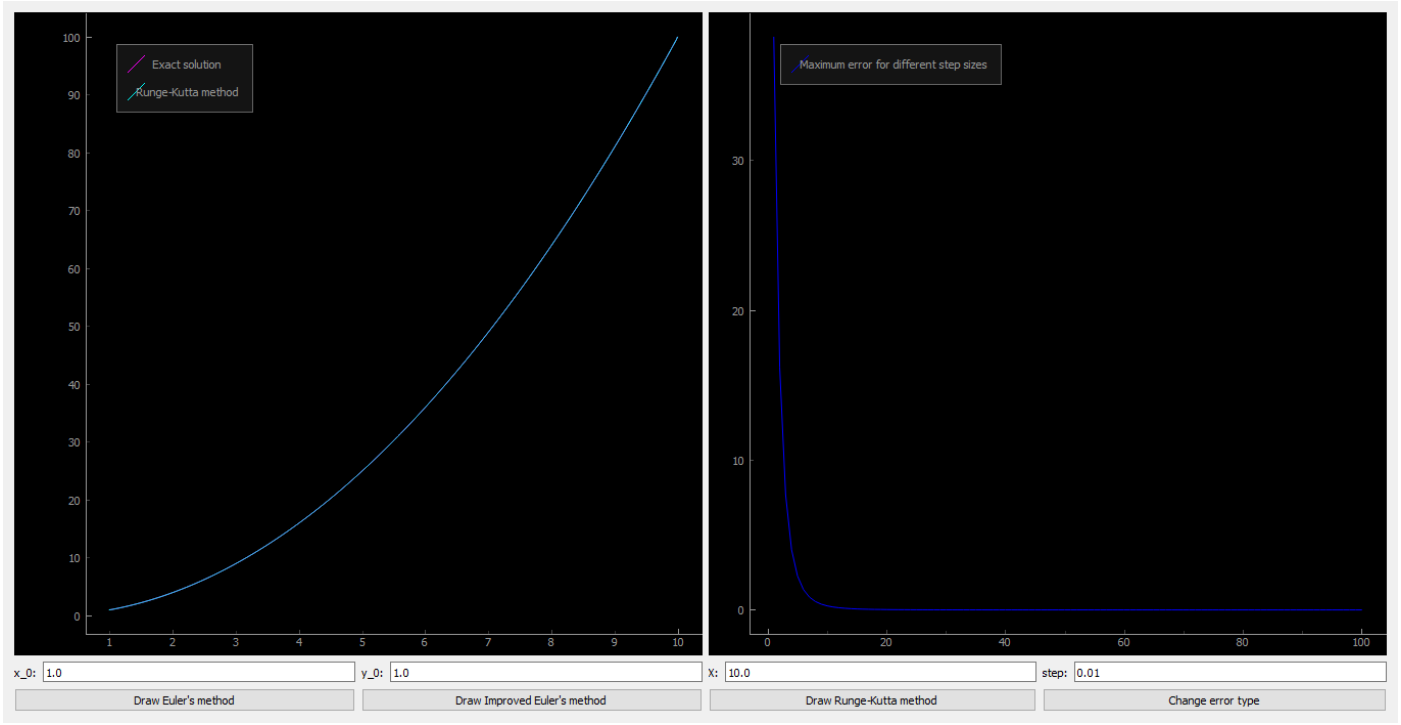
This method uses triple approximation for point trying to predict three derivatives. Here are function plots and error plot for default inputs (plot axes are the same):



Maximum error with default inputs is about 0.000000007. Difference is visible with zoom no less than 3 million times. Here is closer look:



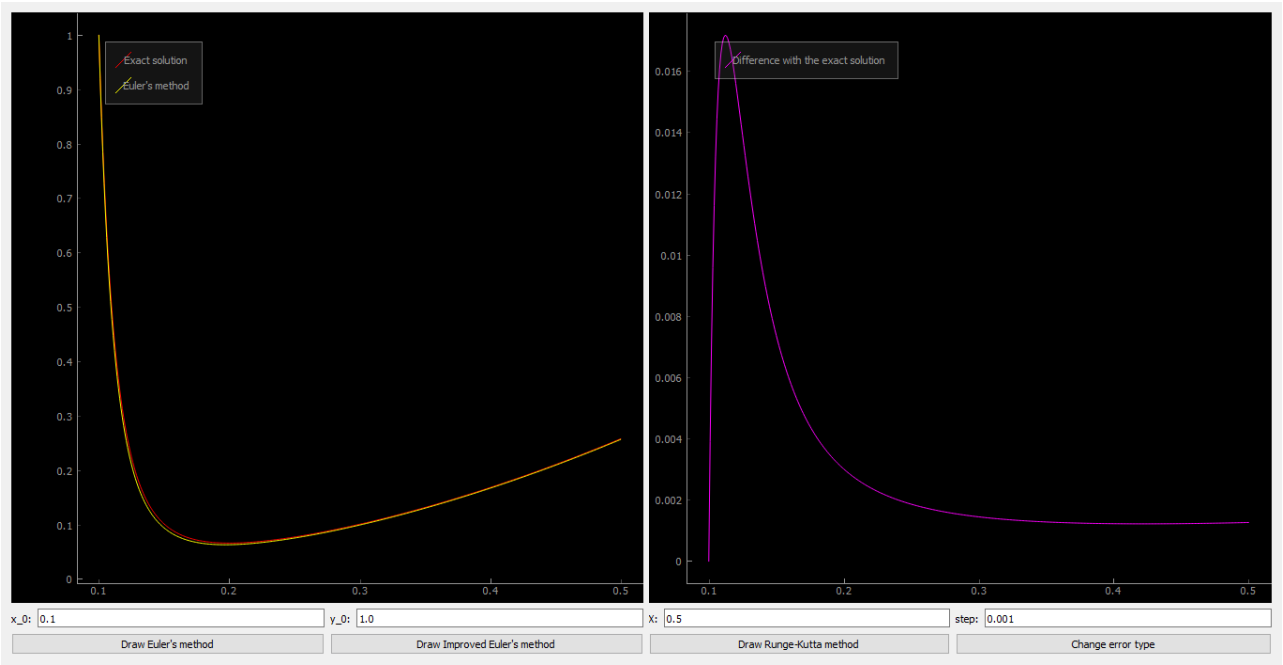
Next you can see maximum error dependence from the step amount (it is decreasing again like for previous methods, but much faster):

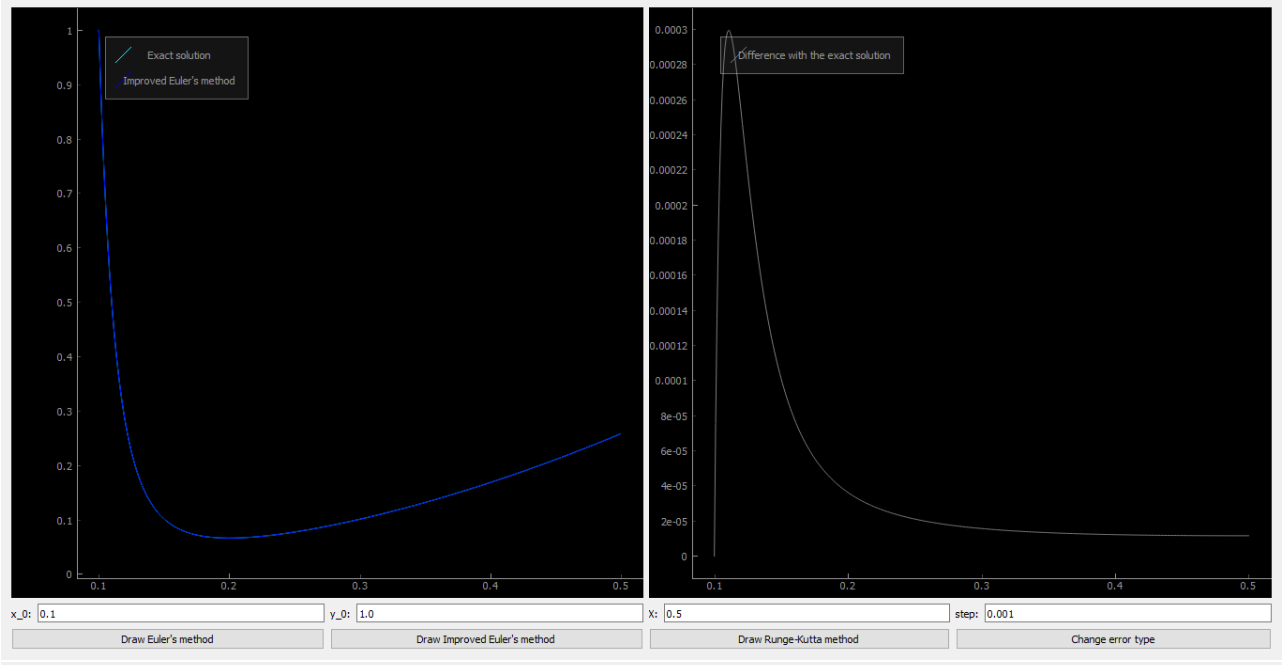


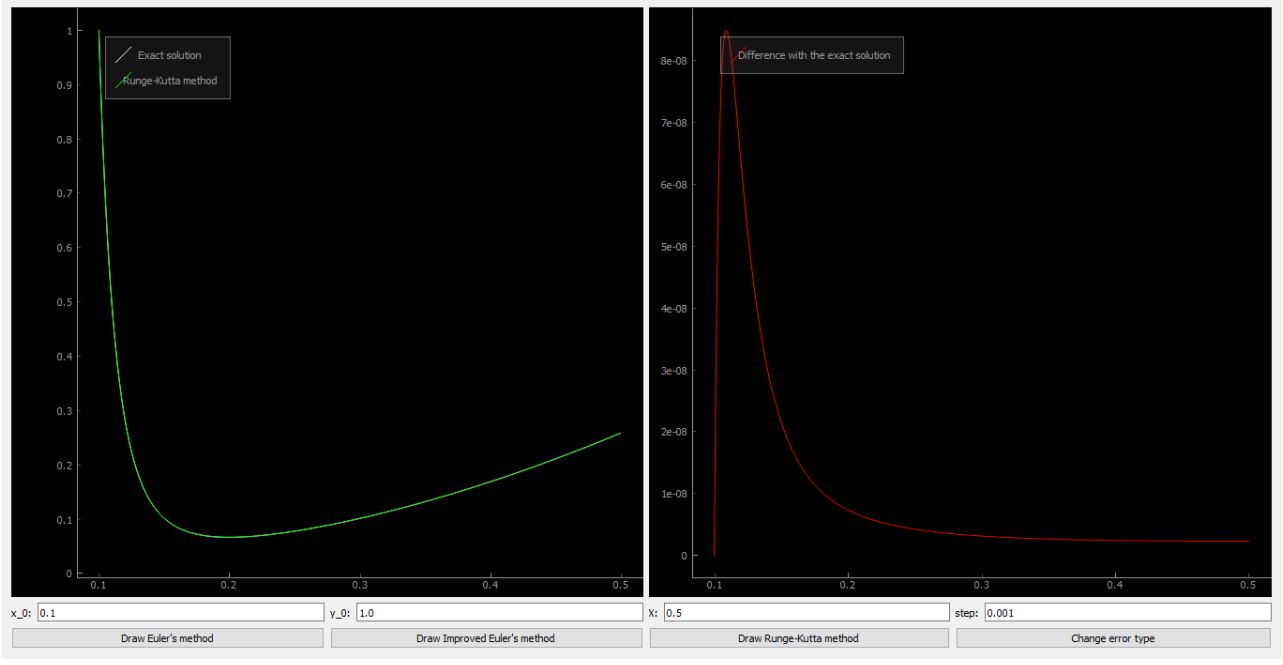
And, finally, code example:

```
def runge_kutta(x_0, y_0, start, end, step):
    xs = [x_0]
    ys = [y_0]
    if start != x_0:
        xs = [start]
        ys = [y(start, c(x_0, y_0))]
    for i, x in enumerate(rational_range(start + step, end + step, step)):
        xs.append(x)
        k1 = f(xs[i], ys[i])
        k2 = f(xs[i] + step / 2, ys[i] + step * k1 / 2)
        k3 = f(xs[i] + step / 2, ys[i] + step * k2 / 2)
        k4 = f(xs[i] + step, ys[i] + step * k3)
        ys.append(ys[i] + step * (k1 + 2 * k2 + 2 * k3 + k4) / 6)
    return xs, ys
```

Other input example







Some notes about the application

To use this application you will need a `Python` interpreter with installed `Pip` package installer. All other required utilities will be installed automatically. To start the application run batch file called `START.BAT` (Windows required, else just run `__main__.py` using your Python interpreter). In the application there are two plotting areas, 4 input fields (for x_0, y_0, X and $step$ of the function) and 4 buttons (3 for plotting each different method and one to change the error plot type). You may scale each plotting area by mouse wheel and change view range by changing options in right-button-click menu.

The application is built using MVC pattern and modules `PyQt5` for GUI and `pyqtgraph` for graph plotting in Qt environment. All the information from the task is stored in `given.py`. Method algorithms and calculation functions are in `utils.py`. MVC Controller subscribes on actions of Qt GUI (button presses and input changes) and changes model's inputs in order to change its state (also stops chosen action if the input is incorrect). MVC View subscribes on the changes in Model (when the state will change, View will act) and draws plots in the corresponding fields on the screen. MVC Model itself is calculating functions at the moment of inputs change and storing them in special variables. `main.py` initializes MVC modules and runs the Qt application. `__main__.py` installs modules (if they're absent (see `modules_inst.py`)) and runs the application itself (`main.py`).

All the calculations are done using `Decimal` data type instead of `float`. It operates on floating-point numbers correctly (for example, sum of ten `0.1` numbers will give `1.0`, not `0.`999…). But it still may have underflow (with about 30 digits after point). Also candidate data type was `Fraction`, which gives absolutely exact result of division, but it requires too much time to calculate functions using our methods.