DIFFERENTIAL EQUATIONS
COMPUTATIONAL PRACTICUM
REPORT

Alexandr Krivonosov
Group 6
Innopolis University 2019

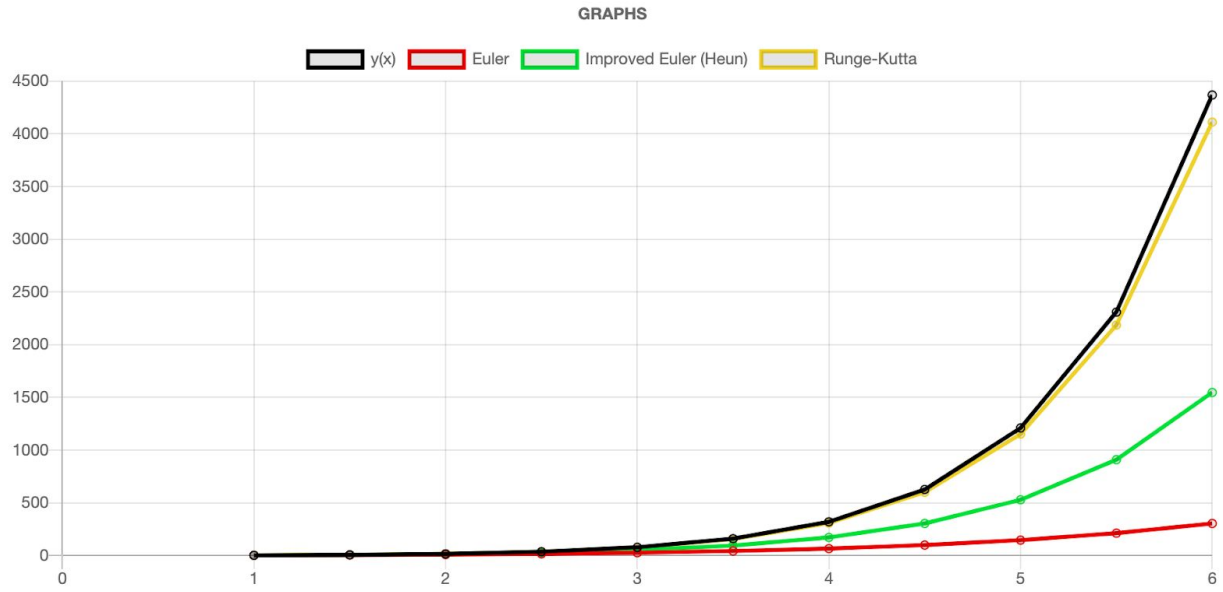
Link To GitHub

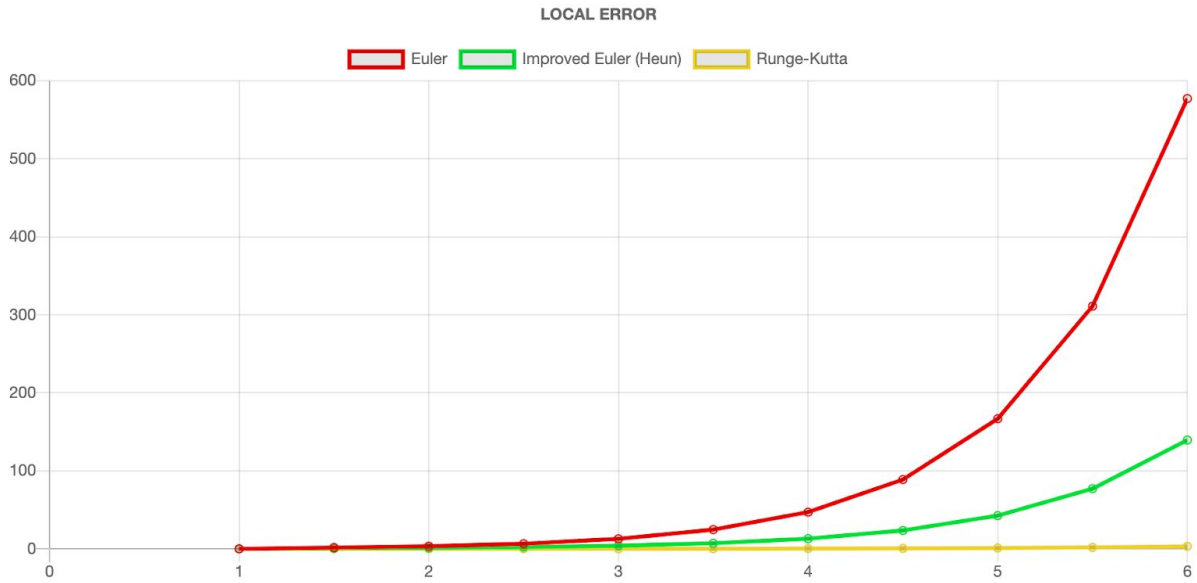
- https://github.com/GneyHabub/DE-Assignment

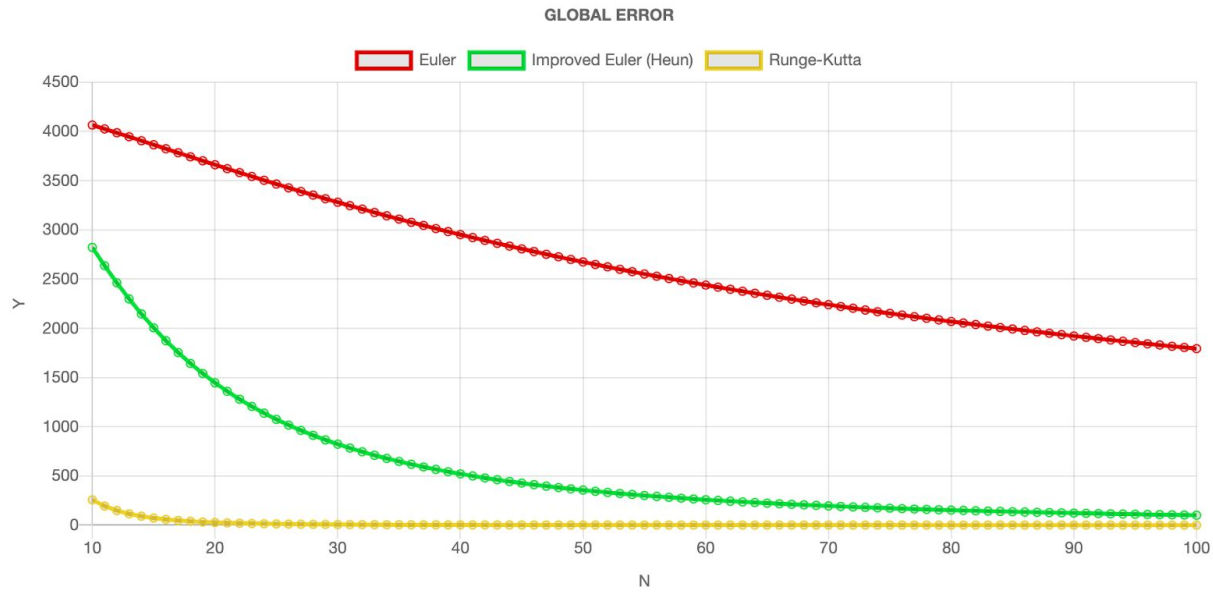
Analytical Solution

$$y' = (1 + y/x)ln((x + y)/x) + y/x;$$
$$v(x) = y(x)/x \Rightarrow dy/dx = x * dv/dx + v;$$
$$x * dv/dx + v = log(1 + v) * (v + 1) + v;$$
$$dv/dx = (log(v+1)(v+1))/x;$$
$$\int dv/(log(v+1)(v+1) = \int dx/x;$$
$$log(log(v+1)) = logx + c;$$
$$v = e^{e^c x} - 1;$$
$$y = x(e^{c1x} - 1),\ where\ c1 = e^c.$$

Results Of The Numerical Solutions (For Given IVP And Step Of Approximation Of 0.5)







Analysis Of The Results

The first chart shows the graph of the analytical solution and three numerical solutions. It is clearly seen that the Euler method is the worst one, Runge-Kutta is the best and Improved Euler (Heun) is in the middle. As the number of points on the chart increases (can be done using *step* field in the App) the Runge-Kutta method's graph almost coincides with the original graph. In fact, the global error in the last point (can be varied using *Max X* field) between the original graph and Runge-Kutta with the step equal to 0.1 (50 points on the chart) is less than 1.

The LOCAL ERROR chart represents, obviously, so-called Local Error, which shows calculation error on each step specifically, regardless of the previous step's error. It is calculated by using the value of the exact solution on each step instead of the previous value of our numerical solution.

The GLOBAL ERROR chart represents the dependency of the quality of the approximations on the number of points. In calculations, there are used *steps* instead of the number of points, but one can be got easily from another using relation: number of points = (max x - min x)/step. So, it is clearly seen from the chart that the more points we have and the less step we have then the better our approximation is.

Analysis Of The Code

The whole project contains 5 files. INDEX.HTML and MAIN.CSS are markup and styles, respectively. The most interesting part is three .JS files with all the computations. The first file is FUNC.JS. It contains all the functions used for computing points. It has the exact solution as `y`, given DE is represented by `y_prime` function. Auxillary function `generate_points` generates an array of integers from the starting point to the end one with a given step. It is used in other functions. Functions `generate_original`, `generate_Euler`, `generate_Improver_Euler`, and `generate_Runge-Kutta` output arrays of objects, containing pairs of points. They all work similarly: generate x-points, compute corresponding y values, put them in the array.

Then there are functions for local error. They work as explained above.

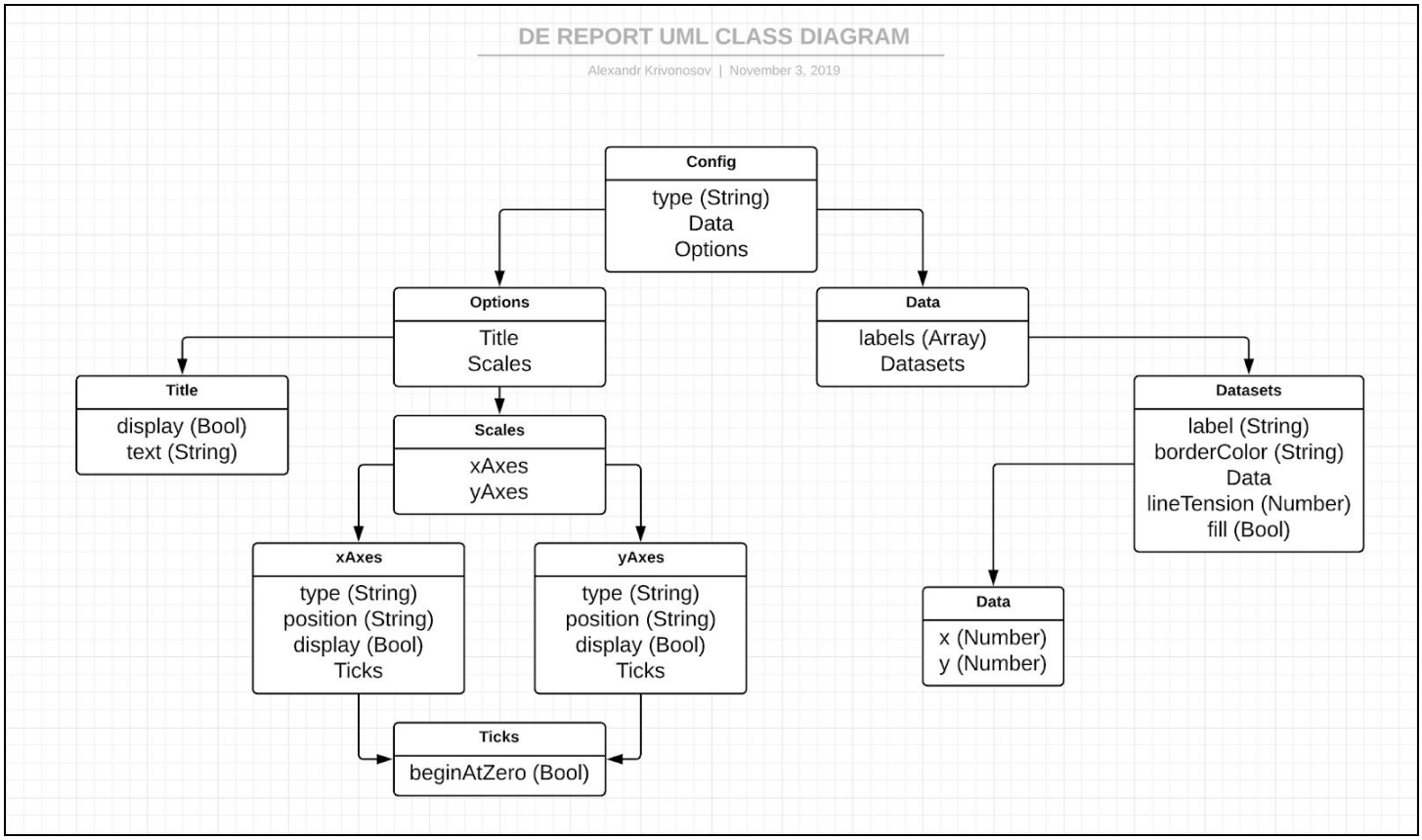
Functions for global error are the most complicated. They recompute the values of corresponding approximations and find the difference between the last points of these recomputed approximations and original function. It happens for all step values between n0 and n1.

The next file, CONFIG.JS, contains all configurations for graphs and also stores the variables for all parameters. The Chart.js library is used in this assignment.

INDEX.JS contains functions for interacting with the page. DOM `ctx1`, `ctx2`, and `ctx3` are the context variables that take corresponding canvases on the page to create graphs on them. `update_graphs` function simply rewrites data to appropriate fields in the configuration file. It is usually done whenever a parameter is changed. The group of `set` functions communicates with the page, takes changed values from appropriate fields, puts them into variables and calls `update_graphs`. `Change_theme` function is simply additional functionality for the page. You can change the theme of the page by clicking the moon icon on the left side of the header. By default, it is dark.

About OOP

Each chart is represented by an instance of the class Chart from the library I used. Using OOP concepts in other parts of the project is redundant. I used a slightly different approach, which is, in my opinion, more appropriate and convenient. *Objects* in JavaScript are something similar to *Structs* in C. It is a grouping of variables or other Objects. This approach is widely used throughout the project, especially for building graphs. The data in CONFIG.JS is represented using Objects. The structure of the `config` variables is shown in the UML diagram on the next page.


DE REPORT UML CLASS DIAGRAM
Alexandr Krivonosov | November 3, 2019
Config
type (String)
Data
Options
Options
Title
Scales
Data
labels (Array)
Datasets
Title
display (Bool)
text (String)
Scales
xAxes
yAxes
Datasets
label (String)
borderColor (String)
Data
lineTension (Number)
fill (Bool)
xAxes
type (String)
position (String)
display (Bool)
Ticks
yAxes
type (String)
position (String)
display (Bool)
Ticks
Data
x (Number)
y (Number)
Ticks
beginAtZero (Bool)