# Report of DE assignment [F18]

## Introduction

**Author**: Ruslan Sabirov
**Group**: B17-02
**GitHub**: github.com/russabirov1998/DE

## Exact solution

```
Exact solution of the Initial Value Problem
```

$y'(x) = 2\,y(x) + e^{x}\,y(x)^2$ — **Bernoulli** equation

$y(x) = -\dfrac{3\,e^{2\,x}}{c_1 + e^{3\,x}}$ — **Solution** of Differential equation

$y(x) = -\dfrac{3\,e^{2\,x}}{e^{3\,x} - 64.4199}$ — Solution of IVP (initial value problem) — **Exact solution**

*x_discont = 1.38847* — Point of **discontinuity** (when denominator of exact solution is equal to 0)

# Structure of the program

`Structure of the program`

Program consist of modules *import part*, *Numerical_mathods* class and *Main part*

## import part

**import** math — *for e constant*

**import** numpy as np — *for working with array of x-coordinates*

**import** plotly — *for plotting*

**import** plotly.graph_objs **as** go — *for creating objects to plot*

## class Numeric_methods()

```
class Numeric_methods():
    h = n = None # h - step, n - number of grid steps
    EPS = 3 * 10 ** (-3) # epsilon
    x_discont = 1.38847 # when denominator of exact solution is equal to 0
    e = math.e # e constant


    def lies_around_discont(self, x) # Whether x lies around discontinuity
    def f(self, x, y) # Given function
    def exact(self, x) # Exact solution of given function
    def __init__(self, x0, y0, X, n) # Main function of class
    def euler_standart(self, x, x0, y0, xf) # Euler method
    def euler_improved(self, x, x0, y0, xf) # Improved Euler method
    def runge_kutta(self, x, x0, y0, xf) # Runge-Kutta method
```

## main part

Creating an object of class Numveric_methods with arguments *x0*, *y0*, *X*, *n*

# Description of methods

Description of each method (there are three methods) with excerpts of code and some comment that explains what the code does.

```
# Euler method
def euler_standart(self, x, x0, y0, xf):
    h = self.h
    f = self.f
    y = [0] * len(x)
    y[0] = y0

    for i in range(1, len(x)):
        if self.lies_around_discont(x[i]): # current point lies around discontinuity
            y[i] = None
            continue
        if len(y) > 1 and y[i - 1] is None: # previous point lies around discontinuity
            y[i] = self.exact(x[i])
            continue

        y[i] = y[i - 1] + h * f(x[i - 1], y[i - 1])
    return y
```

```
# Improved Euler method
def euler_improved(self, x, x0, y0, xf):
    h = self.h
    f = self.f
    y = [0] * len(x)
    y[0] = y0

    for i in range(1, len(x)):
        if self.lies_around_discont(x[i]): # current point lies around discontinuity
            y[i] = None
            continue
        if len(y) > 1 and y[i - 1] is None: # previous point lies around discontinuity
            y[i] = self.exact(x[i])
            continue

        delta_y = h * f(x[i - 1] + h / 2, y[i - 1] + h / 2 * f(x[i - 1], y[i - 1])) # augmentation
        y[i] = y[i - 1] + delta_y
    return y
```

```
# Runge-Kutta method
def runge_kutta(self, x, x0, y0, xf):
    h = self.h
    f = self.f
    y = [0] * len(x)
    y[0] = y0

    for i in range(1, len(x)):
        if self.lies_around_discont(x[i]): # current point lies around discontinuity
            y[i] = None
            continue
        if len(y) > 1 and y[i - 1] is None: # previous point lies around discontinuity
            y[i] = self.exact(x[i])
            continue

        # Assigning coordinates of previous point to variables
        x_prev = x[i - 1]
        y_prev = y[i - 1]
        k1 = f(x_prev, y_prev)
        k2 = f(x_prev + h / 2, y_prev + h * k1 / 2)
        k3 = f(x_prev + h / 2, y_prev + h * k2 / 2)
        k4 = f(x_prev + h, y_prev + h * k3)

        delta_y = h / 6 * (k1 + 2 * k2 + 2 * k3 + k4) # augmentation

        y[i] = y[i - 1] + delta_y
    return y
```

# Solution graph

Display of the graphs the program displays (what I should get if I run it, but I would like to see the output without to have to run it - in live grading I can check if the graphs of the report correspond to what you get on your computer)

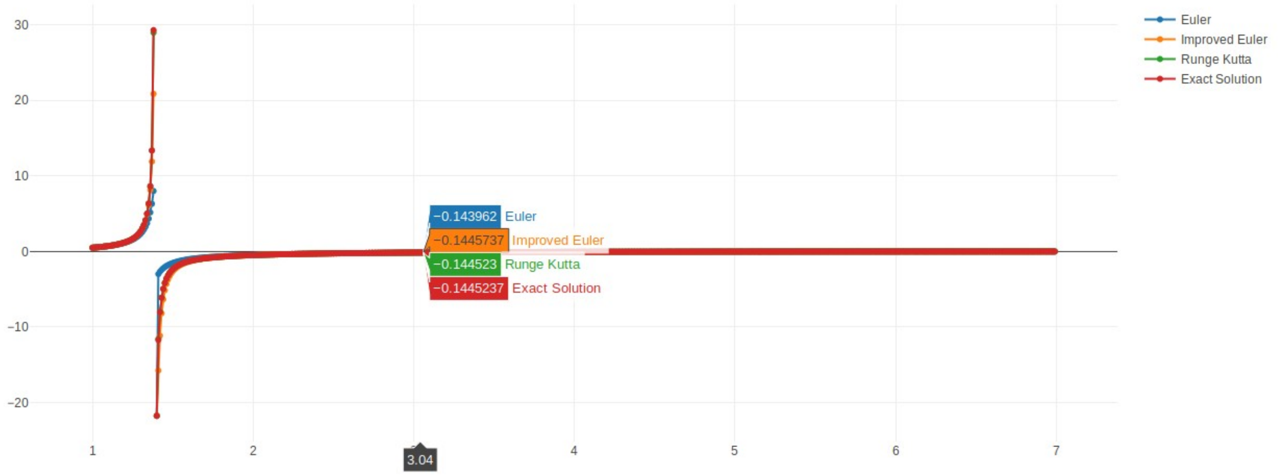


*Illustration 1: X = 7*

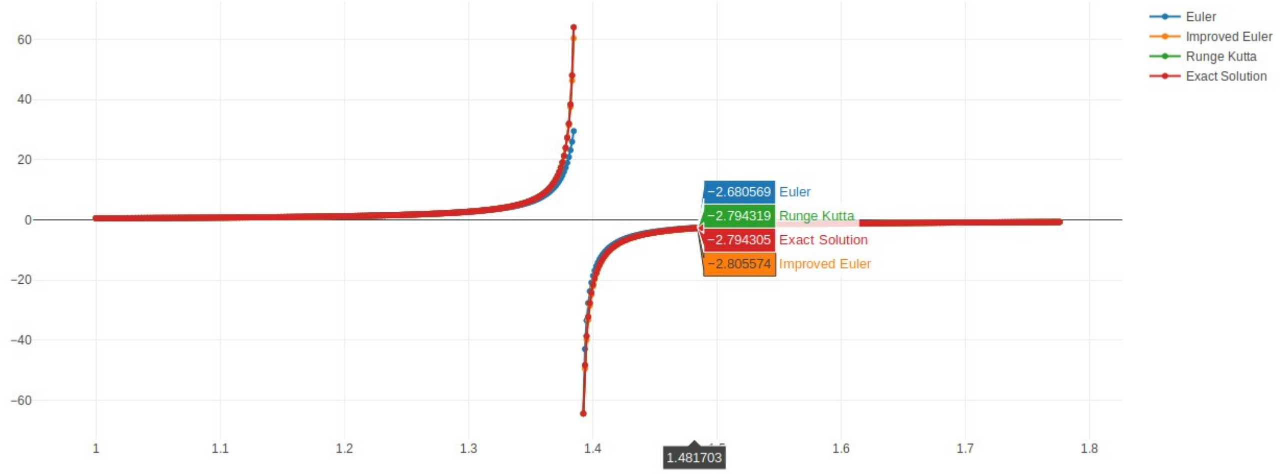


*Illustration 2: X = (x_discont - x0) * 2 + x0 = 1.77694*

# Truncation graph

And, the last but not the least: A graph with three plots that show the global truncation error for each method in function of the number of steps (of course if the number of steps increases, then the error decreases - we want to compare how it decreases for each method and to see which method is the best one).

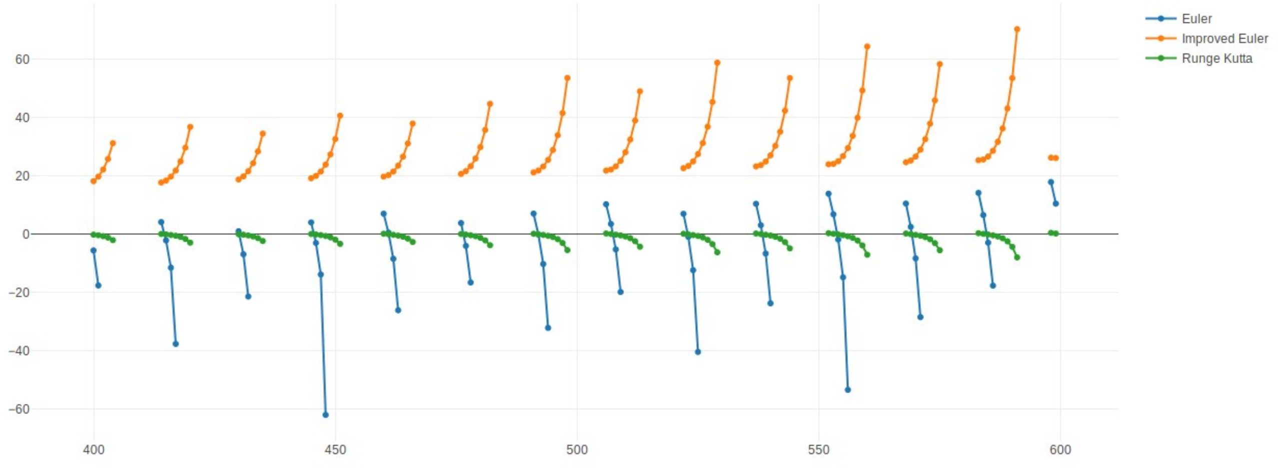


*Illustration 3: Graph of truncation errors over number of steps*

# Code

**GitHub**: [github.com/russabirov1998/DE](github.com/russabirov1998/DE)