Differential equations
Computational practicum

Nikola Novarlic, BS17-03
October 2018

Problem statement:

$y' = 2y^{1/2} + 2y$
$y(0) = 1$
$x \in [0;9]$

Exact solution of the Initial Value Problem

$$\int \frac{dy}{2(\sqrt{y}+y)} = \int dx$$

$$\int \frac{1}{2} \frac{dy}{(\sqrt{y}+1)\sqrt{(y)}} = x + c$$

$$u = \sqrt{y}+1$$

$$dy = 2\sqrt{y}\ du$$

$$\frac{1}{2}\ 2\int \frac{1}{u}\ du = x + c$$

$$\ln(u) + c = x + c$$

$$\ln(\sqrt{y}+1) = x + c$$

$$\sqrt{y}+1 = e^{x + c}$$

$$\sqrt{y} = e^{x + c} - 1$$

$$y = (e^{x + c} - 1)^2$$

github: https://github.com/greendate/DE-Numerical-methods

By applying initial values we get:

$$(e^{c-1})^2 = 1$$

$$e^{c-1} = 1$$

$$e^{c} = 2$$

$$c = \log(2)$$

And exact solution is:

$$y = (2e^{x-1})^2$$

There are no points of discontinuity in given range.

Implementation

I have chosen python language. With 2 libraries imported, `matplotlib` for plotting graphs and `tkinter` for GUI.

Program structure

Besides numerical methods and exact solutions, which are straight forward implemented, we have routines for computing approximating errors (plotting difference between method and numerical methods) and investigating convergence. For simplicity *f(x,y)* and *y(x)* are implemented as functions.

Euler method

```
def euler_method(x0, y0, b, n):
        h = (b - x0) / n
        x = []
        for i in range(n):
                x.append(x0 + i * h)
        y = [y0]
        for i in range(1, n):
                y.append(y[i - 1] + h * f(x[i - 1], y[i - 1]))
        return y
```

Improved Euler method

```
def improved_euler_method(x0, y0, b, n):
	h = (b - x0) / n
	x = []
	for i in range(n):
		x.append(x0 + i * h)
	y = [y0]
	for i in range(1, n):
		k1 = f(x[i - 1], y[i - 1])
		k2 = f(x[i - 1] + h, y[i - 1] + h * k1)
		y.append(y[i - 1] + (h / 2) * (k1 + k2))
	return y
```

Runge-Kutta method

```
def runge_kutta_method(x0, y0, b, n):
	h = (b - x0) / n
	x = []
	for i in range(n):
		x.append(x0 + i * h)
	y = [y0]
	for i in range(1, n):
		k1 = f(x[i - 1], y[i - 1])
		k2 = f(x[i - 1] + h / 2, y[i - 1] + (h / 2) * k1)
		k3 = f(x[i - 1] + h / 2, y[i - 1] + (h / 2) * k2)
		k4 = f(x[i - 1] + h, y[i - 1] + h * k3)
		y.append(y[i - 1] + (h / 6) * (k1 + 2 * k2 + 2 * k3 + k4))
	return y
```

Each method gets same values for x and produces approximations on that points, *Runge-Kutta method* is most accurate, while typical Euler method gives the least precise results.

***** for k_{i1}, k_{i2}, k_{i3}, k_{i4} were used as variables inside loop instead of using arrays to store them

github: https://github.com/greendate/DE-Numerical-methods

Convergence

Here we check if maximal approximation error of given method using really large N tends to 0, that is, nearest integer of all values have to be 0. With 100 000 as number of grid steps we will get that only Runge-Kutta method is convergent.

Plotting

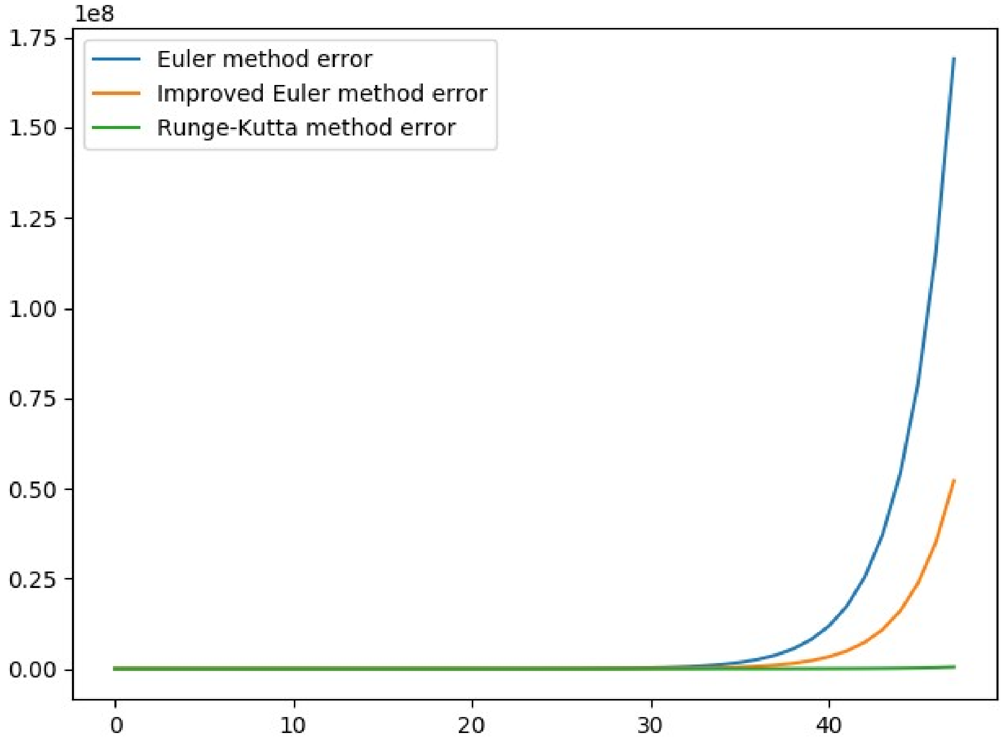


Figure 1: approximation errors

github: https://github.com/greendate/DE-Numerical-methods

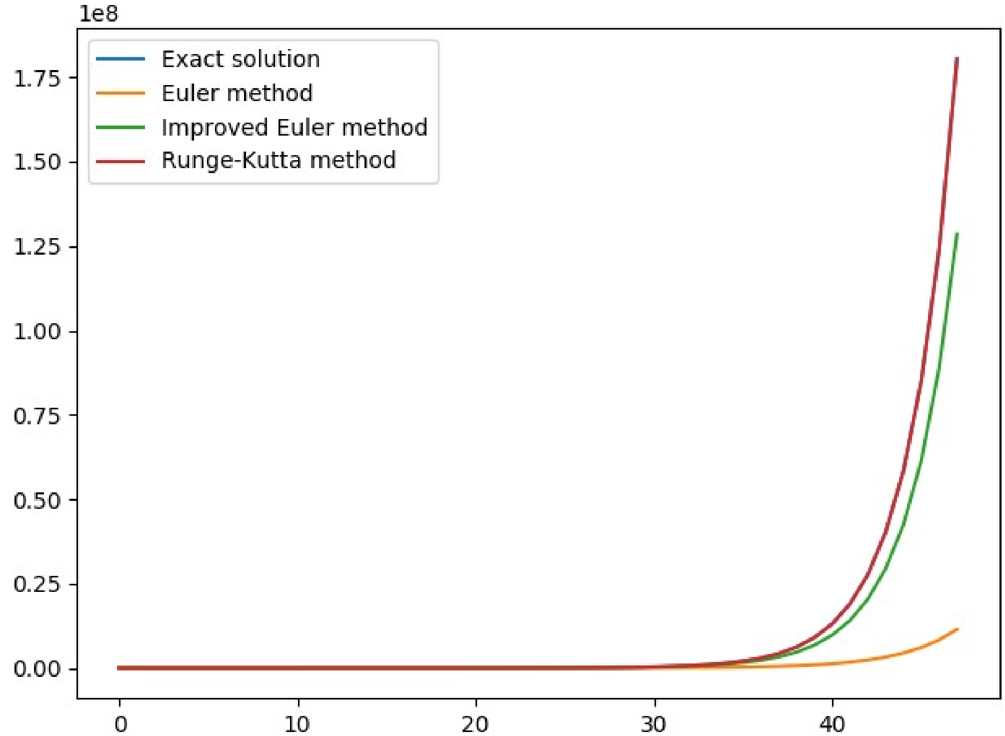


Figure 2 : Exact solution against numerical methods (plots for exact solution and Runge-Kutta method coincide)

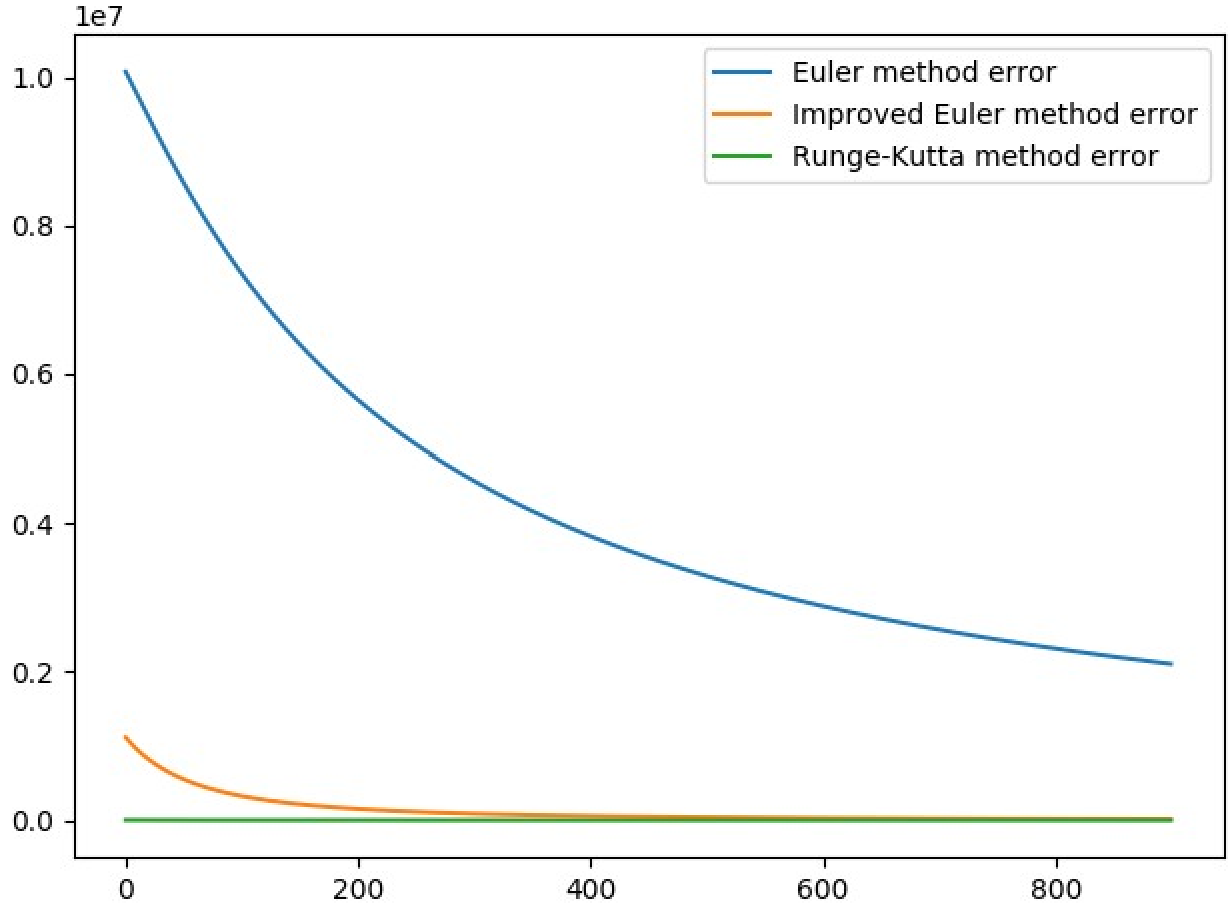
Following functions were used for them:

```
def plotErrors(error1, error2, error3):
    err_line1, = plt.plot(error1)
    err_line2, = plt.plot(error2)
    err_line3, = plt.plot(error3)
    plt2.legend([err_line1, err_line2, err_line3],
                ['Euler method error', 'Improved Euler method error', 'Runge-Kutta method
                  error'])
    plt2.show()



def plotMethods(exact, euler, improved, runge_kutta):

    line1, = plt.plot(exact)
    line2, = plt.plot(euler)
    line3, = plt.plot(improved)
    line4, = plt.plot(runge_kutta)
    plt.legend([line1, line2, line3, line4],
               ['Exact solution', 'Euler method', 'Improved Euler method', 'Runge-Kutta
                 method'])
    plt.show()
```

Here we can see that Runge-Kutta gives best approximation.

github: https://github.com/greendate/DE-Numerical-methods

Error as a function of number of steps



There is also graph that shows that shows how error decreases while number of step increases. User is allowed to choose interval where he wants to examine the graph. Method works as following: compute exact solution and numerical method with different values of N from range given, take average error for numerical methods at each step and plot that graph.

github: https://github.com/greendate/DE-Numerical-methods

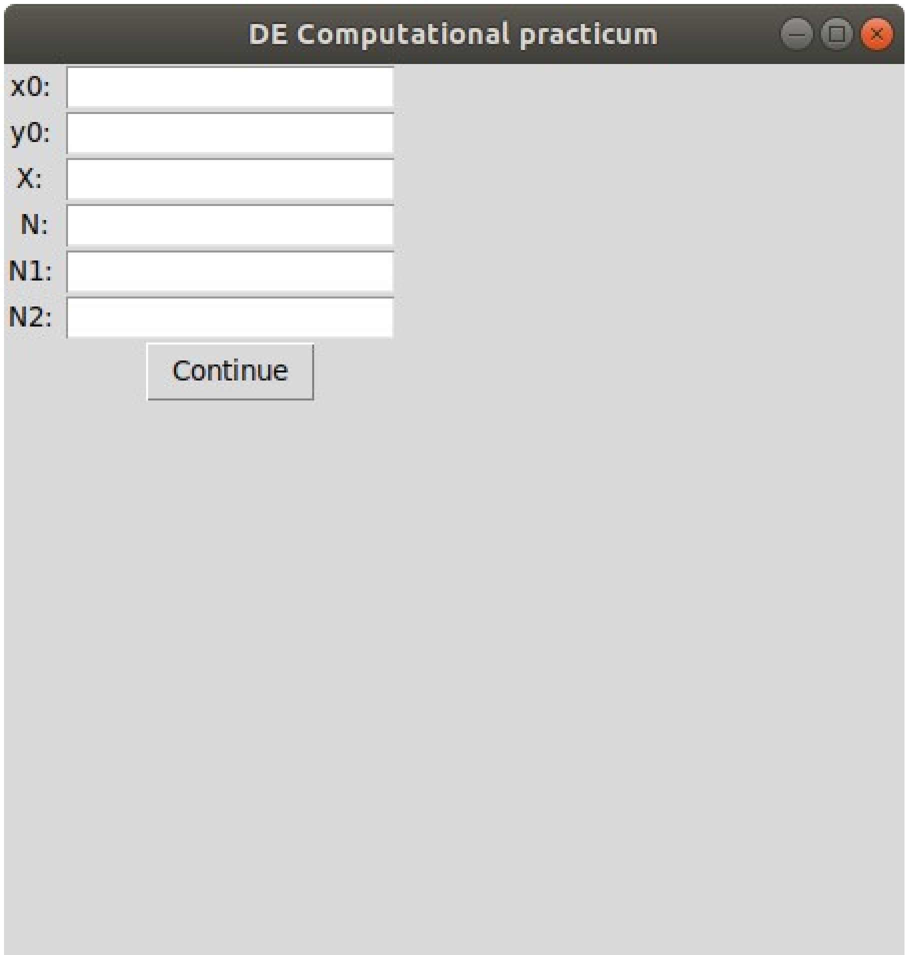
Procedure used for this:

```python
def plot_total_errors():

    total_errors_e = []  # Euler
    total_errors_ie = []  # Improved Euler
    total_errors_rk = []  # Runge-Kutta

    for i in range(n1, n2):
        # values of exact solution and numerical methods when we use i computational steps
        exact_i = exact_solution(x0, y0, b, i)
        euler_i = euler_method(x0, y0, b, i)
        improved_i = improved_euler_method(x0, y0, b, i)
        runge_kutta_i = runge_kutta_method(x0, y0, b, i)

        # compute their errors
        error1_i = compute_error(exact_i, euler_i)
        error2_i = compute_error(exact_i, improved_i)
        error3_i = compute_error(exact_i, runge_kutta_i)

        # append average values of errors
        total_errors_e.append(sum(error1_i) / i)
        total_errors_ie.append(sum(error2_i) / i)
        total_errors_rk.append(sum(error3_i) / I)

    plotErrors(total_errors_e, total_errors_ie, total_errors_rk)
```

* note that same plotting function we used for errors

github: https://github.com/greendate/DE-Numerical-methods

Graphic user interface



github: https://github.com/greendate/DE-Numerical-methods