DE assignment on numerical methods
Rinat Babichev
BS17 - 08

Problem:

y'=-2y+4x
y(0)=0, x ∈ [0,3]

Solution:

dy/dx + 2y = 4x
dy/dx + 2y = 0 => y = C*e^(-2x),
Change C with a function u(x):
u' * e^(-2x) = 4x
du = 4x*e^(2x) * dx,

$\int$ du = $\int$ 4x*e^(2x) * dx,

u = 2x*e^(2x) - e^(2x) + c
Substitute back and get:
y=(2x*e^2x - e^2x + c)*e^(-2x) ,
y=2x-1+c*e^(-2x),
y(0)=0 => c=1
So solution to ivp is y=2x-1+e^(-2x)

There are no points of discontinuity

Implementation:

I have chosen Python programming language and used following libraries:
Math - for computing mathematical functions
Matplotlib - for plotting graphs
Numpy - for constructing array of values in some range with float step

Program:

Each method receives following arguments:

f - callable function of x and y such that y'=f(x,y),
xs - array of x-values
y0 - given ivp value for xs[0]
h - step

Euler method:
```python
def euler(f, xs, y0, h):
    ys = [y0]
    for i in range(1, xs.size):
        ys += [ys[i - 1] + h * f(xs[i - 1], ys[i - 1])]
    return ys
```

Improved Euler:
```python
def improved_euler(f, xs, y0, h):
    ys = [y0]
    for i in range(1, xs.size):
        k1 = f(xs[i - 1], ys[i - 1])
        k2 = ys[i - 1] + h * k1
        ys += [ys[i - 1] + h / 2 * (k1 + f(xs[i], k2))]
    return ys
```

Runge-Kutta:

```python
def runge_khutta(f, xs, y0, h):
    ys = [y0]
    for i in range(1, xs.size):
        k1 = f(xs[i - 1], ys[i - 1])
        k2 = f(xs[i - 1] + h / 2, ys[i - 1] + h * k1 / 2)
        k3 = f(xs[i - 1] + h / 2, ys[i - 1] + h * k2 / 2)
        k4 = f(xs[i - 1] + h, ys[i - 1] + h * k3)
        ys += [ys[i - 1] + h / 6 * (k1 + 2 * k2 + 2 * k3 + k4)]
    return ys
```

- Each method is also wrapped with decorator(@calculate_error), which calculates local error for values got from method and returns it

**Method for plotting each graph:**

```python
def plot(xs,y0,h,ysol):
    tup = (func, xs, y0, h, ysol)
    # draw euler
    y, error_euler = euler(*tup)
    plt.plot(xs, y, 'b', label=euler_)
    # draw exact solution
    plt.plot(xs, ysol, 'r', label=exact_)
    # draw improved euler
    y, error_improved = improved_euler(*tup)
    plt.plot(xs, y, 'y', label=improved_)
    # draw runge-khutta
    y, error_runge = runge_khutta(*tup)
    plt.plot(xs, y, 'g', label=runge_)
    plt.legend()
    # plot errors
    print_errors(xs,  error_euler, error_improved, error_runge)
```

Method for plotting Global error graph:

```python
def global_error(x0, x1, y0):
    error_euler = []
    error_improved_euler = []
    error_runge_kutta = []
    N = range(10, 2000)
    for n in N:
        h = (x1 - x0) / n
        xs = numpy.arange(x0, x1 + h, h)
        ysol = solution(xs)
        tup = (func, xs, y0, h, ysol)
        _, error1 = euler(*tup)
        _, error2 = improved_euler(*tup)
        _, error3 = runge_khutta(*tup)
        error_euler.append(max(error1))
        error_improved_euler.append(max(error2))
        error_runge_kutta.append(max(error3))
        print_errors(N, error_euler, error_improved_euler, error_runge_kutta)
```
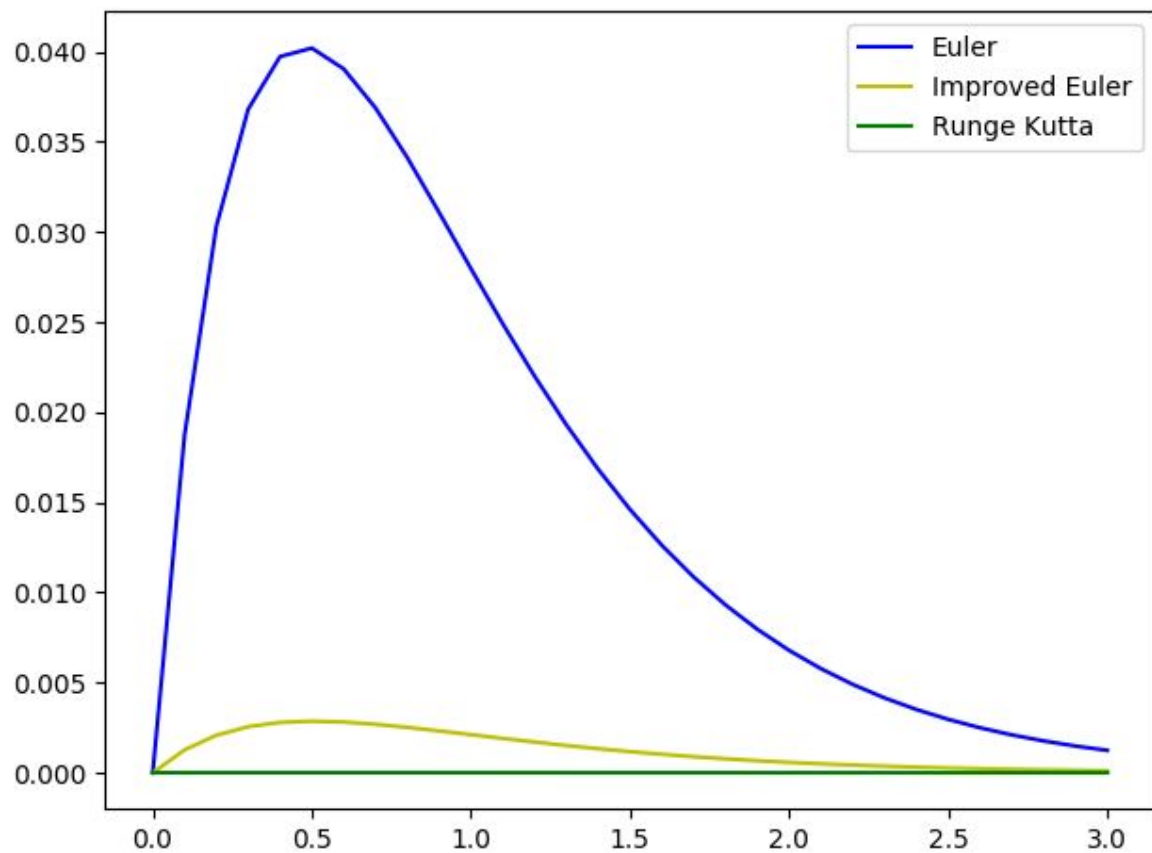
Results

*Graphs for each method with step h=0.1*
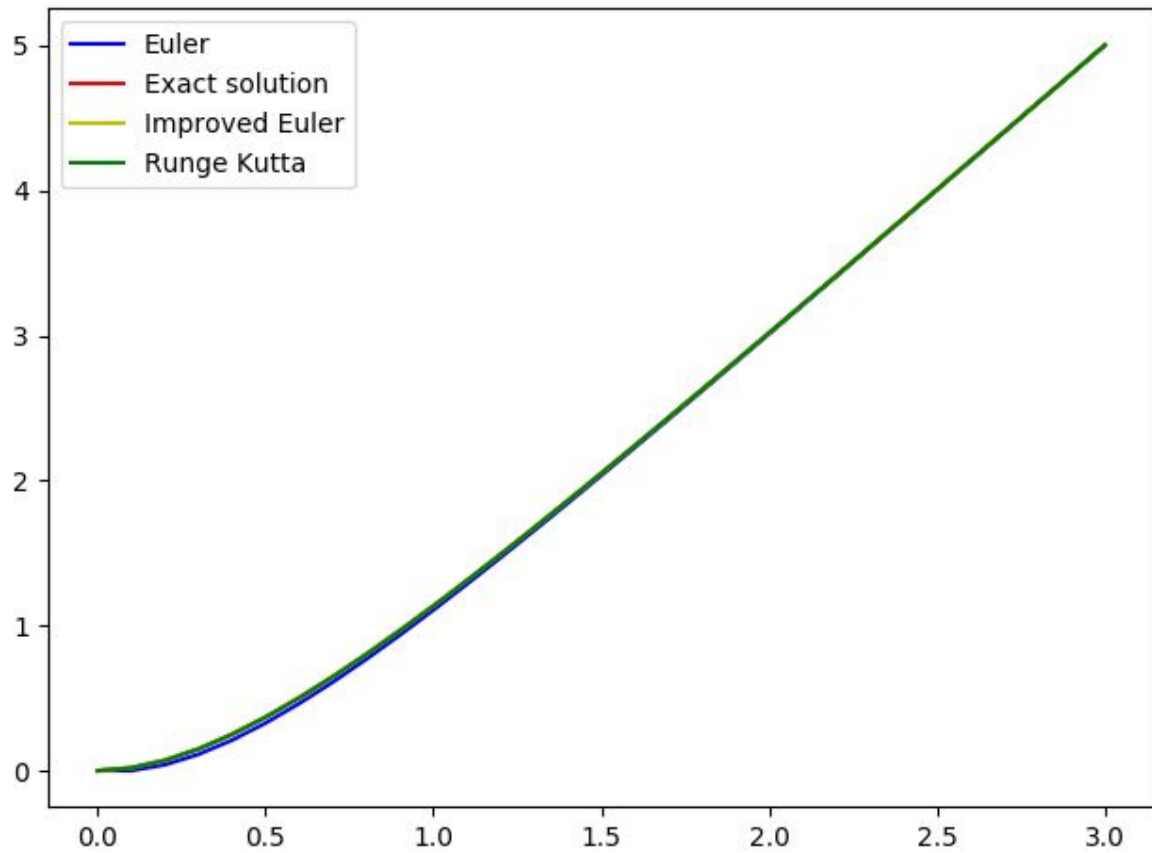
- As we see all three plots are very close to the graph of an exact solution

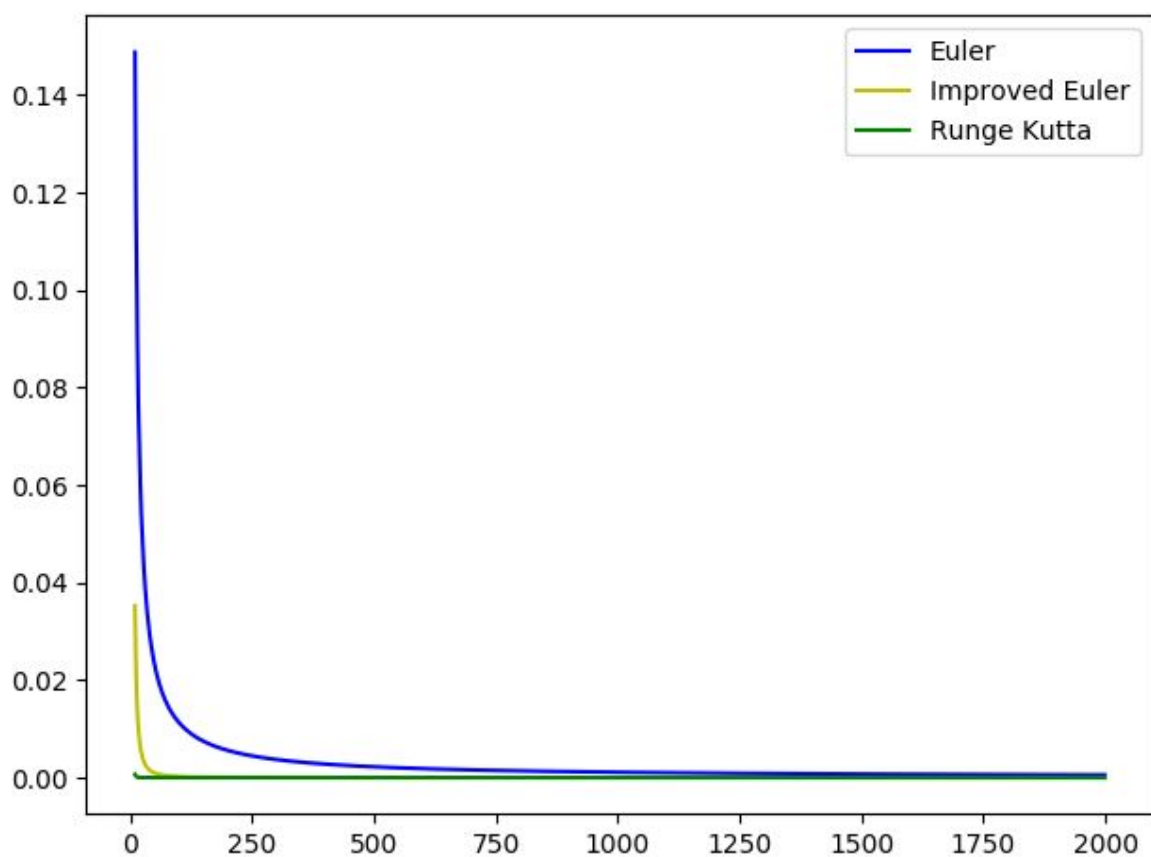*Graph of local errors for each method*

- In the beginning of calculations each method shows increasing local error near starting point location, however, as the number of calculated points increases accuracy increases too

Runge Kutta shows the best approximation, it's local error graph closer:
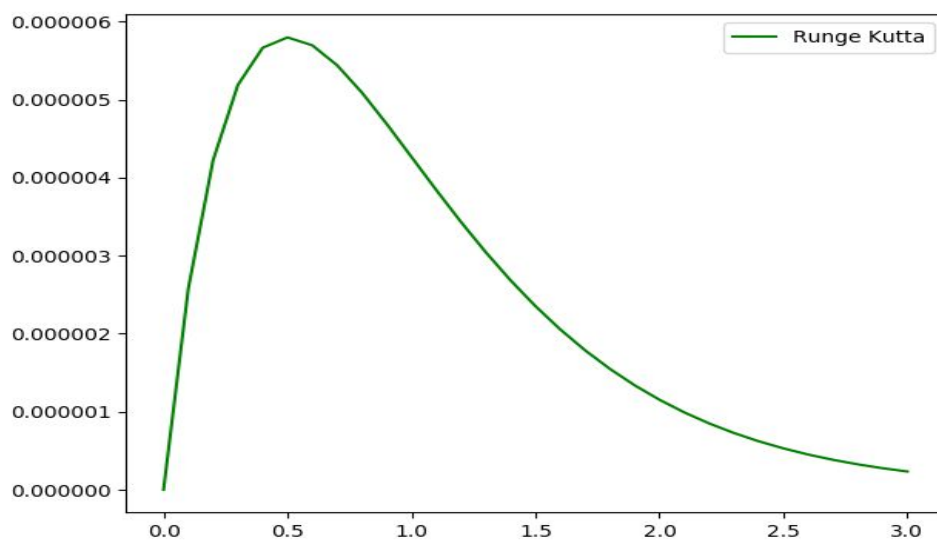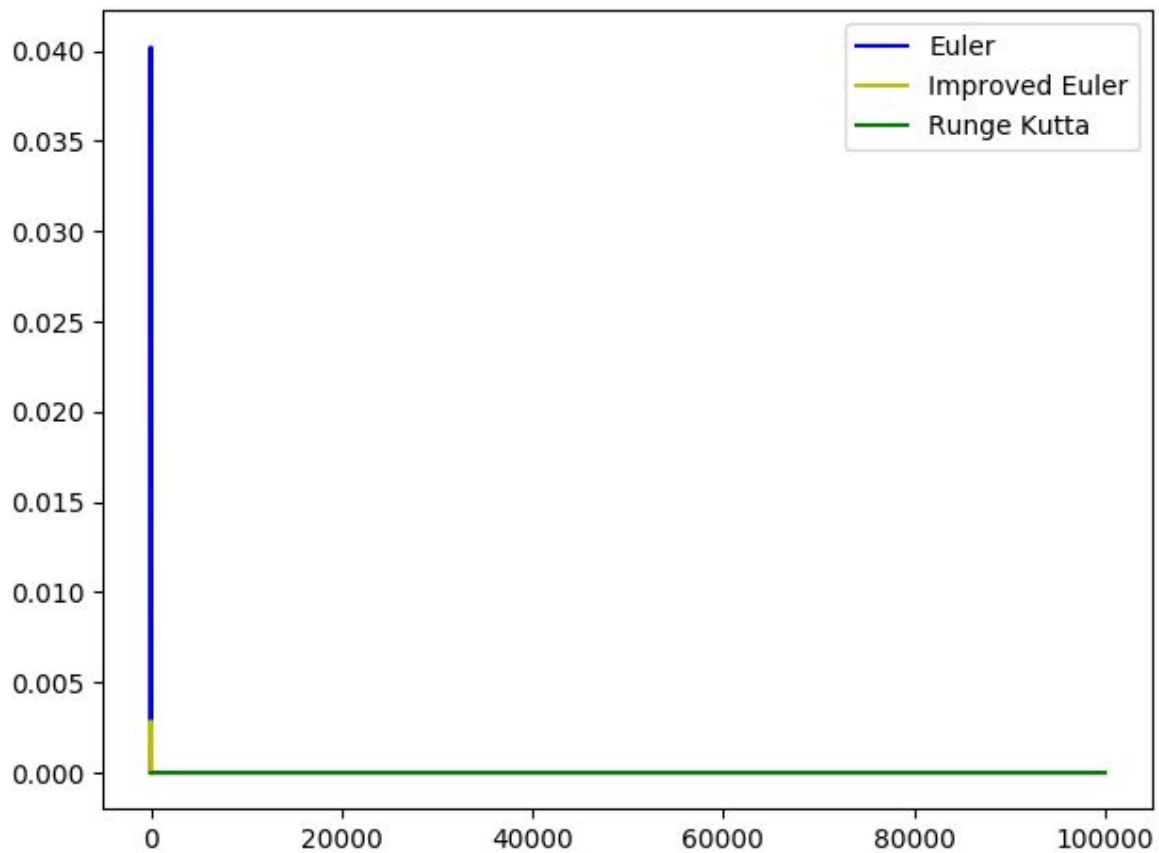

Runge Kutta local error graph

Graph of dependence of max local error on the number of approximated points

● Accuracy directly proportional to the number of calculated points

Convergence

We check that maximal approximation error of a method tends to zero with really large N, every approximation tends to zero so all methods converge

Graph of local error for each method for xn = 100 000