Differential Equation

Find Exact

Find exact solution:

$$\begin{cases} y' = (1 + \frac{y}{x})ln(\frac{(x+y)}{x}) + \frac{y}{x} \\ y(1) = 2 \\ x \in (1, 6) \end{cases}$$

$x \neq 0$

First order nonlinear ordinary differential equation

Let's do substitution $z = 1 + \frac{y}{x}$, than $z' = \frac{y'x + x'y}{x^2}$, $y = xz - x$, $y' = \frac{x^2z' + xz - x}{x}$

The equation becomes:

$\frac{x^2z' + xz - x}{x} = zln(z) + z - 1 \quad \rightarrow \quad z'x = zln(z)$

Convert equation into separable form:

$\frac{dz}{dx}x = zln(z) \rightarrow \frac{dz}{zlnz} = \frac{dx}{x}$

Integrate both part

$\int \frac{dz}{zlnz} = \int \frac{dx}{x} \rightarrow ln(ln(z)) = ln(x) + ln(c) \rightarrow ln(z) = xc$

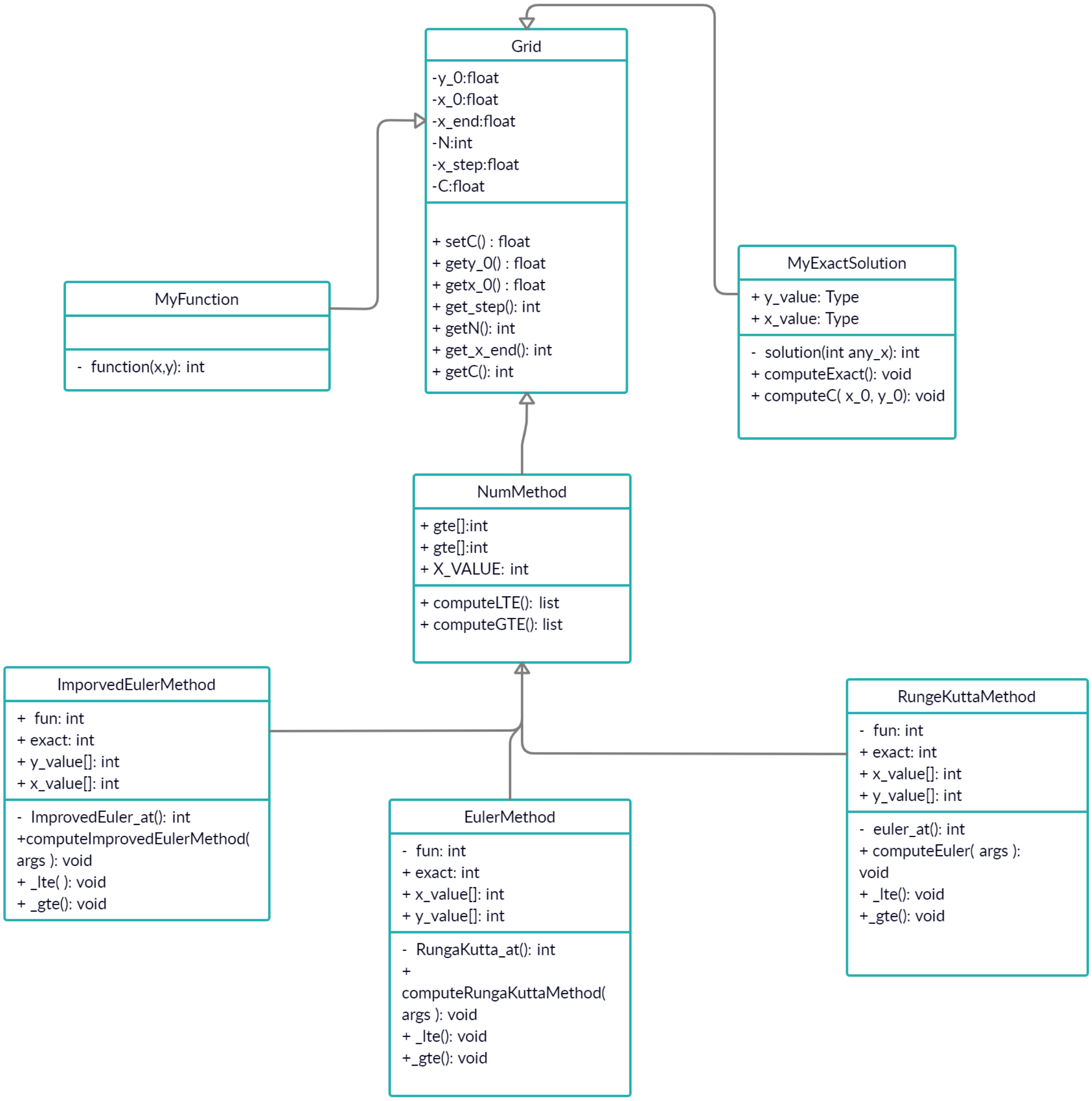
Substitute back to original equation

$ln(1 + \frac{y}{x}) = xc \rightarrow 1 + \frac{y}{x} = e^{xc} \rightarrow y = (e^{xc} - 1)x$

Solve IVP: if $x = 1$ and $y = 2$ than $2 = e^c - 1 \rightarrow c = ln(3)$

Exact solution: $y = (e^{xln(3)} - 1)x$

UML - diagram of class



The main class of the source code consists of four classes: MyFunction, GRID, NumMethod, MyExactSolution and ImprovedEulerMethod, RungeKuttaMethod, EulerMethod. Classes NumMethod and MyExactSolution use the class Function in their implementation, while class NumMethod extend the class Solutions and Solutions extends Grid. The class MyFunction contains the given differential equation. The class Grid contains the step, points of the x axes and y points for all numerical methods and exact solution. The class MyExactSolution finds y values for all methods. The class

NumMethod finds all types of errors and the dependence of errors on the number of grid cells.

```
class EulerMethod(NumMethod):
    def __init__(self, y_0, x_0, x_end, N):
        super().__init__(y_0, x_0, x_end, N)
        self.fun = MyFunction()
        self.exact = MyExactSolution(y_0, x_0, x_end, N)
        self.x_value = [self.x_0]
        self.y_value = [self.y_0]

    def euler_at(self, x, y):
        return y + self.get_step() * self.fun.function(x, y)

    def computeEuler(self):
        for i in range(1, self.getN()+1):
            x_i = self.x_value[i - 1]
            y_i = self.y_value[i - 1]
            self.y_value.append(y_i + self.get_step() * self.fun.function(x_i, y_i))
            self.x_value.append(x_i + self.get_step())

    def _lte(self):
        for i in range(1, self.getN()+1):
            self.computeLTE(self.exact.solution(self.x_value[i]), self.euler_at(self.x_value[i-1], self.exact.solution(self.x_value[i-1])))

    def _gte(self):
        for i in range(1, self.getN()+1):
            self.computeGTE(self.exact.solution(self.x_value[i-1]), self.y_value[i-1])
```

```
class RungeKuttaMethod(NumMethod):

    def __init__(self, y_0, x_0, x_end, N):
        super().__init__(y_0, x_0, x_end, N)
        self.fun = MyFunction()
        self.exact = MyExactSolution(y_0, x_0, x_end, N)
        self.x_value = [self.x_0]
        self.y_value = [self.y_0]

    def RungaKutta_at(self, x, y):
        k1 = self.fun.function(x, y)
        k2 = self.fun.function(x+self.get_step() / 2, y + self.get_step() / 2 * k1)
        k3 = self.fun.function(x+self.get_step() / 2, y + self.get_step() / 2 * k2)
        k4 = self.fun.function(x+self.get_step(), y + self.get_step() * k3)
        return y + self.get_step() / 6 * (k1 + 2 * k2 + 2 * k3 + k4)


    def computeRungaKuttaMethod(self):
        for i in range(1, self.getN()+1):
            k1 = self.fun.function(self.x_value[i-1], self.y_value[i - 1])
            k2 = self.fun.function(self.x_value[i-1]+self.get_step()/2, self.y_value[i - 1] + self.get_step() / 2 * k1)
            k3 = self.fun.function(self.x_value[i-1]+self.get_step()/2, self.y_value[i - 1] + self.get_step() / 2 * k2)
            k4 = self.fun.function(self.x_value[i-1]+self.get_step(), self.y_value[i - 1] + self.get_step() * k3)
            self.y_value.append( self.y_value[i-1]+self.get_step()/6*(k1+2*k2+2*k3+k4) )
            self.x_value.append(  self.x_value[i-1]+self.get_step() )
```

```python
class ImprovedEulerMethod(NumMethod):

    def __init__(self, y_0, x_0, x_end, N):
        super().__init__(y_0, x_0, x_end, N)
        self.fun = MyFunction()
        self.exact = MyExactSolution(y_0, x_0, x_end, N)
        self.y_value = [self.y_0]
        self.x_value = [self.x_0]

    def ImprovedEuler_at(self, x, y):
        EulerMethod = y + self.get_step() * self.fun.function(x, y)
        return y + (self.get_step() / 2) * (
                    self.fun.function(x, y) + self.fun.function(x + self.get_step(), EulerMethod))

    def computeImprovedEulerMethod(self):
        for i in range(1, self.getN()+1):
            x_i = self.x_value[i - 1]
            y_i = self.y_value[i - 1]
            EulerMethod = y_i + self.get_step() * self.fun.function(x_i, y_i)
            self.y_value.append(y_i + (self.get_step() / 2) * (self.fun.function(x_i, y_i) + self.fun.function(self.x_value[i-1] + self.get_ste
            self.x_value.append(self.x_value[i-1] + self.get_step())

    def _lte(self):
        for i in range(1, self.getN()+1):
            self.computeLTE(self.exact.solution(self.x_value[i]), self.ImprovedEuler_at(self.x_value[i-1], self.exact.solution(self.x_value[i-1
```

```python
from GRID import GRID


class NumMethod(GRID):
    def __init__(self, y_0, x_0, x_end, N):
        super().__init__(y_0, x_0, x_end, N)
        self.lte = [0]
        self.gte = [0]
        self.X_VALUE = [x_0]
        for i in range(1, N+1):
            self.X_VALUE.append((self.X_VALUE[i-1])+self.get_step())

    def computeLTE(self, ExactSolution, Approximate):
        self.lte.append(abs(ExactSolution - Approximate))

    def computeGTE(self, Exact, Method):
        self.gte.append(abs(Exact - Method))
```

```
class GRID:
    def __init__(self, y_0, x_0, x_end, N):
        self._step = (x_end-x_0)/N
        self.N = N
        self.y_0 = y_0
        self.x_0 = x_0
        self.x_end = x_end
        self.C = math.log(3)

    def setC(self, value):
        self.C = value

    def gety_0(self):
        return self.y_0

    def getx_0(self):
        return self.x_0

    def get_step(self):
        return self._step

    def getN(self):
        return self.N
```

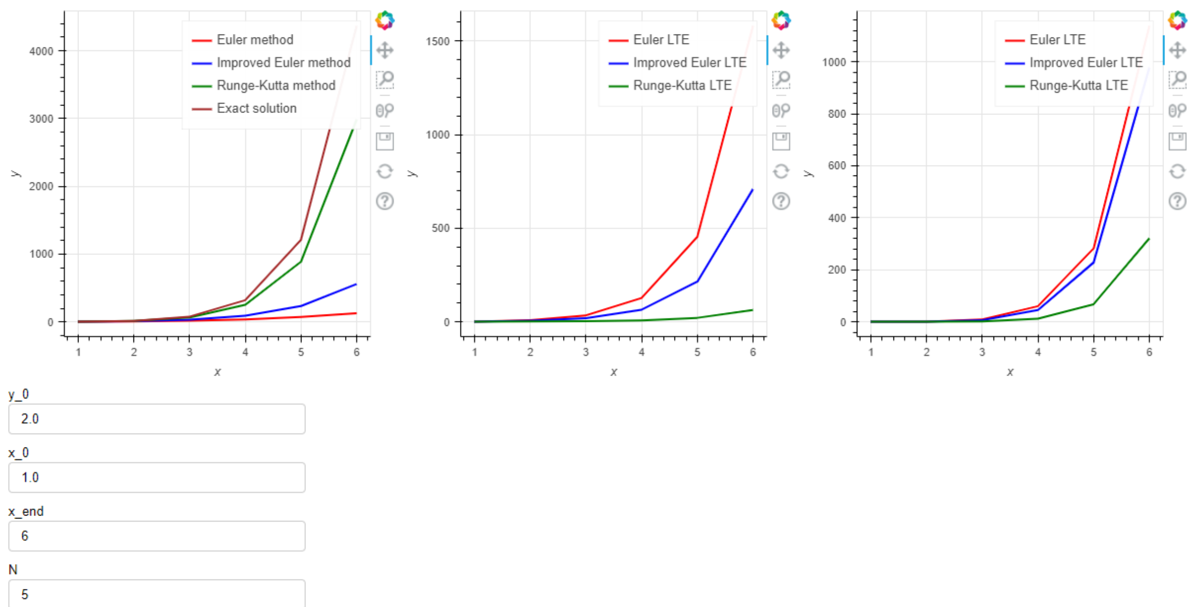
```
from GRID import GRID


class MyFunction():
    def __init__(self):
        return

    def function(self, x, y):
        return (1+y/x)*math.log((x+y)/x)+y/x
```

There is the first tab of the application below. It shows the graph of exact and numerical solutions, the graph of local truncation errors for numerical solutions and the graph of global

truncation errors for numerical solutions. It is possible to hide solutions on the plots. Values x0,

y0, X and N can be changed.

From this graphs we can see that the best approximation gives the Runge-Kutta method.

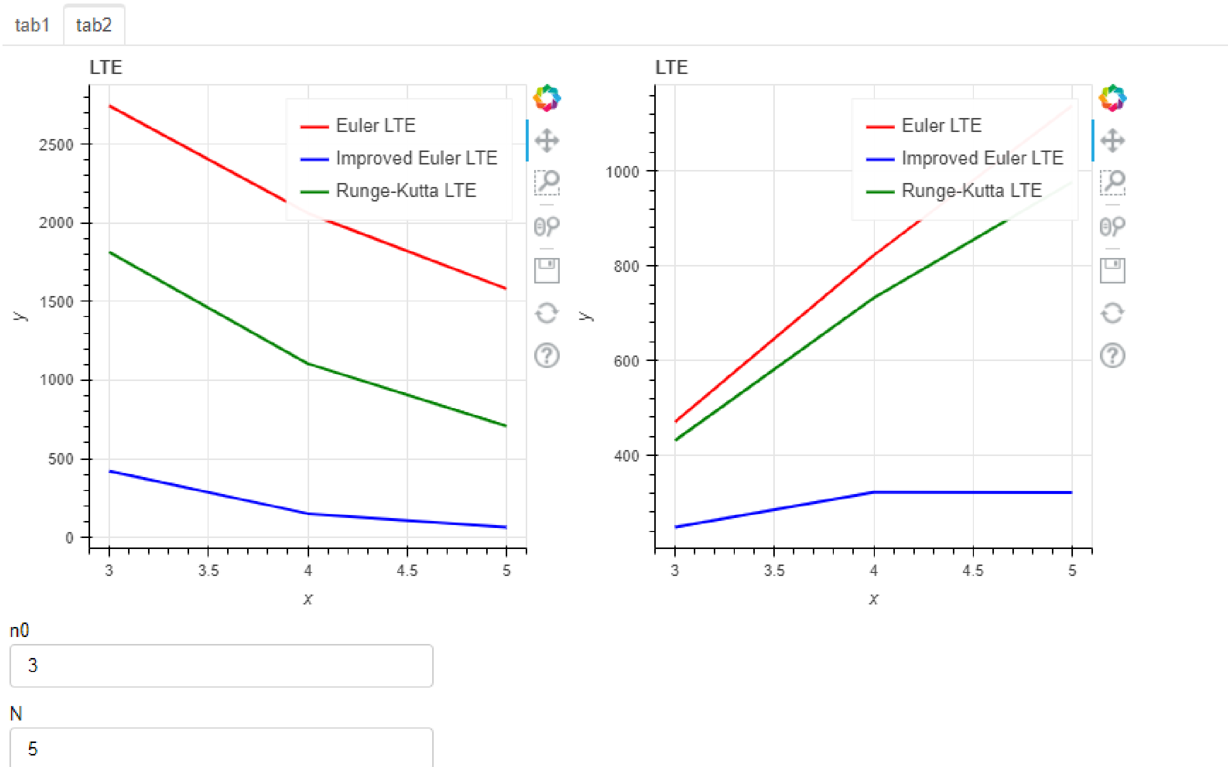


There is the second tab of the application below. It shows the dependence of errors from the

number of grid cells for local and global truncation errors. It is possible to hide lines on graphs.

Values n0 and N can be changed. Values x0, y0, X and N are taken from the first tab.

From these graphs one can notice that the more the number of grid cells, the less the error


tab1
tab2
LTE
Euler LTE
Improved Euler LTE
Runge-Kutta LTE
2500
2000
1500
1000
500
0
y
3
3.5
4
4.5
5
x
LTE
Euler LTE
Improved Euler LTE
Runge-Kutta LTE
1000
800
600
400
y
3
3.5
4
4.5
5
x
n0
3
N
5