Differential Equations: Computational Practicum

Roman Nabiullin

October 22, 2020

Variant 5

Part 1.

Solve the IVP $y' = \frac{y}{x} + x \cdot \cos x$, $y(x_0) = y_0$ analytically.

Solution.

Let us notice that we are dealing with a **Bernoulli equation**.

Let us make a substitution $y = x \cdot u(x)$ where x is a non-trivial solution for the complementary equation $y_c' - \frac{y_c}{x} = 0$.

Then $x \cdot u' = x \cdot \cos x$.

Assuming that $x \neq 0$ which is not a solution, we have $u' = \cos x$.

Thus, $u(x) = \sin x + C$, and

$$y(x) = x \cdot \sin x + Cx$$

is the most general solution for the source equation on $(-\infty, 0) \cup (0, +\infty)$.

As we know that $y(x_0) = y_0$, we can find C.

$$C = \frac{y_0 - x_0 \cdot \sin x_0}{x_0}$$

Finally, we have

$$y(x) = x \cdot \sin x + x \cdot \frac{y_0 - x_0 \cdot \sin x_0}{x_0}$$

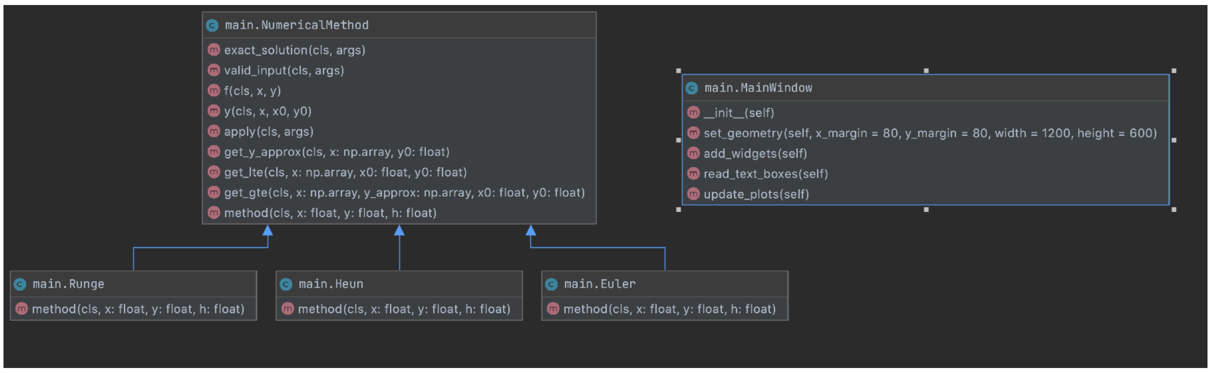
The interval of validity is $(-\infty, 0)$ if $x_0 < 0$, or $(0, +\infty)$ if $x_0 > 0$.

Part 2.

Provide a brief code overview and UML diagram.

Solution.

It was decided to use Python programming language (version 3.8).


main.NumericalMethod
exact_solution(cls, args)
valid_input(cls, args)
f(cls, x, y)
y(cls, x, x0, y0)
apply(cls, args)
get_y_approx(cls, x: np.array, y0: float)
get_lte(cls, x: np.array, x0: float, y0: float)
get_gte(cls, x: np.array, y_approx: np.array, x0: float, y0: float)
method(cls, x: float, y: float, h: float)
main.MainWindow
__init__(self)
set_geometry(self, x_margin = 80, y_margin = 80, width = 1200, height = 600)
add_widgets(self)
read_text_boxes(self)
update_plots(self)
main.Runge
method(cls, x: float, y: float, h: float)
main.Heun
method(cls, x: float, y: float, h: float)
main.Euler
method(cls, x: float, y: float, h: float)

We have an abstract class *NumericalMethod* that contain all the necessary functions for applying numerical methods, for instance

```
@classmethod
def y(cls, x, x0, y0):
    return x*np.sin(x) + x*(y0-x0*np.sin(x0))/x0


@classmethod
def f(cls, x, y):
    return y/x + x*np.cos(x)


@classmethod
def get_y_approx(cls, x: np.array, y0: float) -> np.array:
    y_approx = np.full(x.shape[0], y0, dtype=float)

    for i in range(1, x.shape[0]):
        y_approx[i] = cls.method(x[i-1], y_approx[i-1], x[1]-x[0])

    return y_approx
```

However, this one is abstract

```
@classmethod
def method(cls, x: float, y: float, h: float):
    pass
```

Each numerical method Euler's, Improved Euler's (aka Heun's) and Runge-Kutta overrides this function like this

```
class Runge(NumericalMethod):
    @classmethod
    def method(cls, x: float, y: float, h: float):
        k1 = cls.f(x, y)
        k2 = cls.f(x + h/2, y + k1*h/2)
        k3 = cls.f(x + h/2, y + k2*h/2)
        k4 = cls.f(x + h, y + k3*h)

        return y + (k1 + 2*k2 + 2*k3 + k4)*h/6
```

This approach works since the algorithm for each method is the same. The only thing that changes is `method()`

$$y_{approx}[i] = \text{method}(x[i-1], y_{approx}[i-1], h)$$

As for the GUI, it was decided to use PyQt5.

```
def set_geometry(self,x_margin=80,y_margin=80,width=1200,height=600):
    self.setWindowTitle("Numerical methods")
    self.setGeometry(x_margin, y_margin, width, height)
    self.setMinimumSize(QSize(width, height))
    self.setMaximumSize(QSize(width, height))

def add_widgets(self):
    # Grid for widgets
    self.grid = QGridLayout()
    self.grid.setSpacing(10)

    # Button to plot
    self.button_trigger = QPushButton(text = "Apply Methods")
    self.button_trigger.clicked.connect(self.read_text_boxes)
    ...
```

One may find the complete code together with more detailed explanation in the comments in .py files here.

Part 3.

Review the GUI.

Solution.

As for the parameters,

[**x0**, **X**] is the interval for 'x'

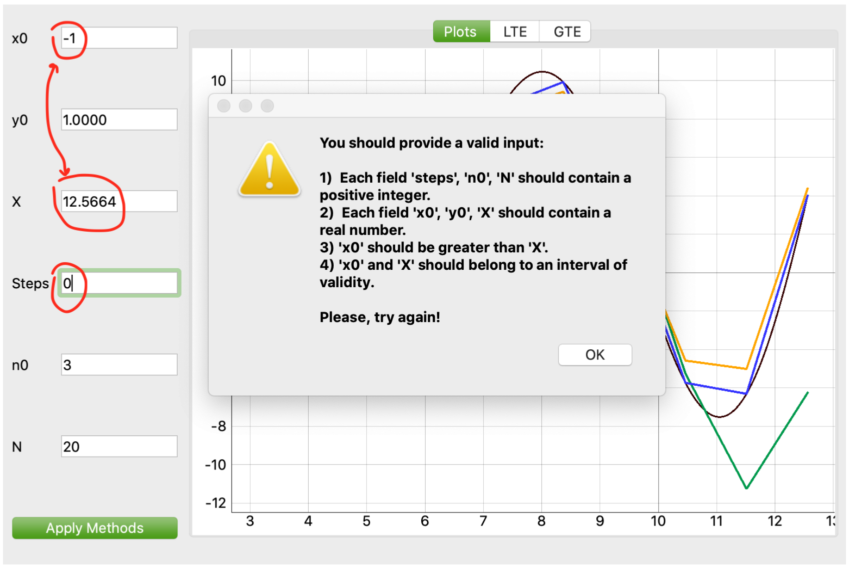
y0 = y(x0)

Steps = number steps for the approximation

n0 and **N** - are parameters for the third plot. They stand for the initial and final number of steps respectively.

After filling the input fields, one should press the button "Apply Methods" and then observe the results. It is possible to switch between the plots.

One should provide the input properly. Otherwise, they will see the following message



You should provide a valid input:

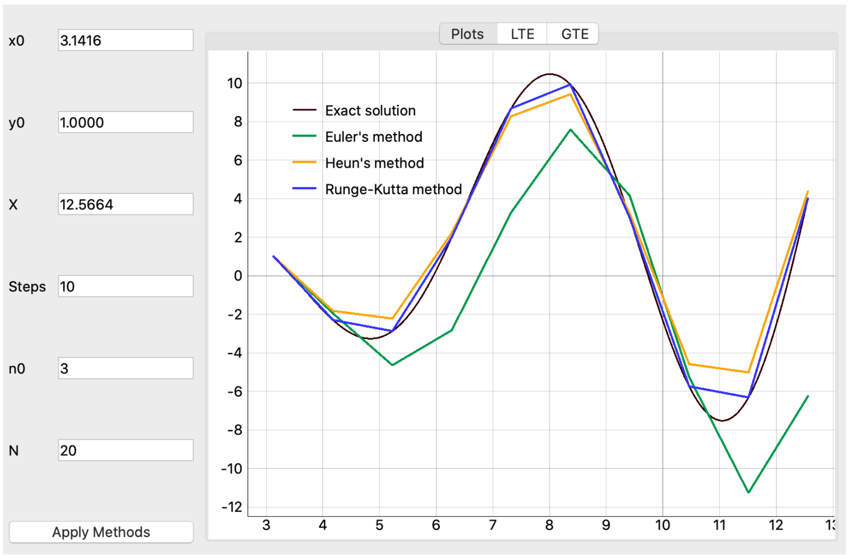
1) Each field 'steps', 'n0', 'N' should contain a positive integer.
2) Each field 'x0', 'y0', 'X' should contain a real number.
3) 'x0' should be greater than 'X'.
4) 'x0' and 'X' should belong to an interval of validity.

Please, try again!

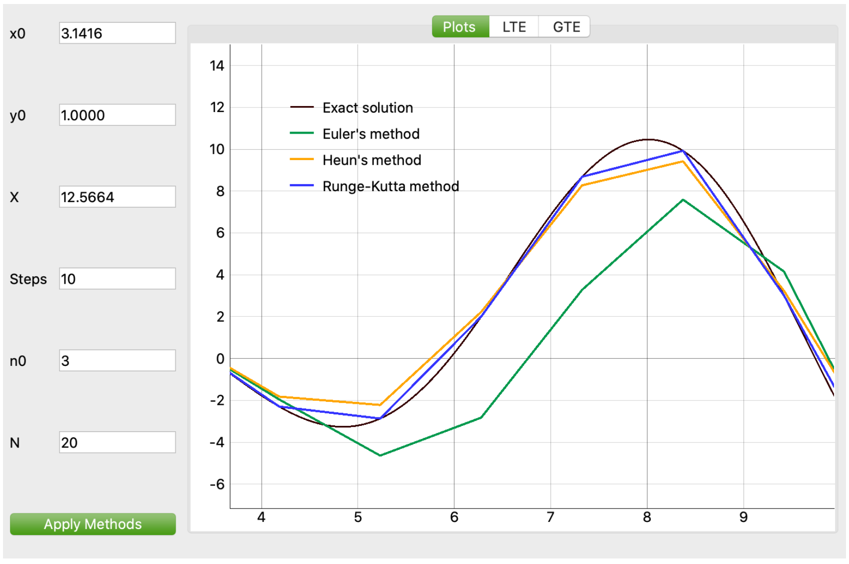
OK

Initially, everything is calculated for the default values. Most of them were provided.

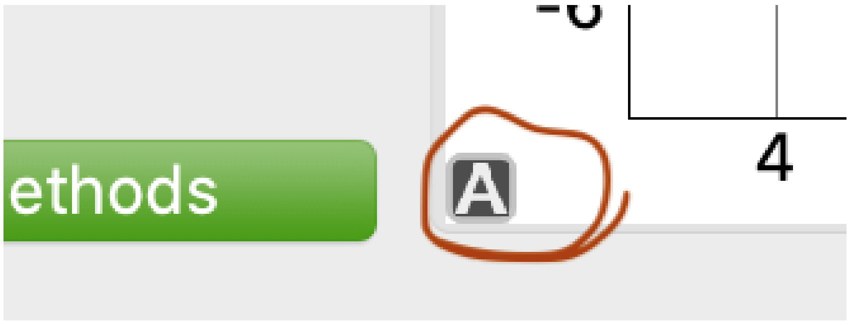
Below, one may see a window of the GUI that contains input fields, a number of required plots and the button "Apply Methods".


x0 3.1416
y0 1.0000
X 12.5664
Steps 10
n0 3
N 20

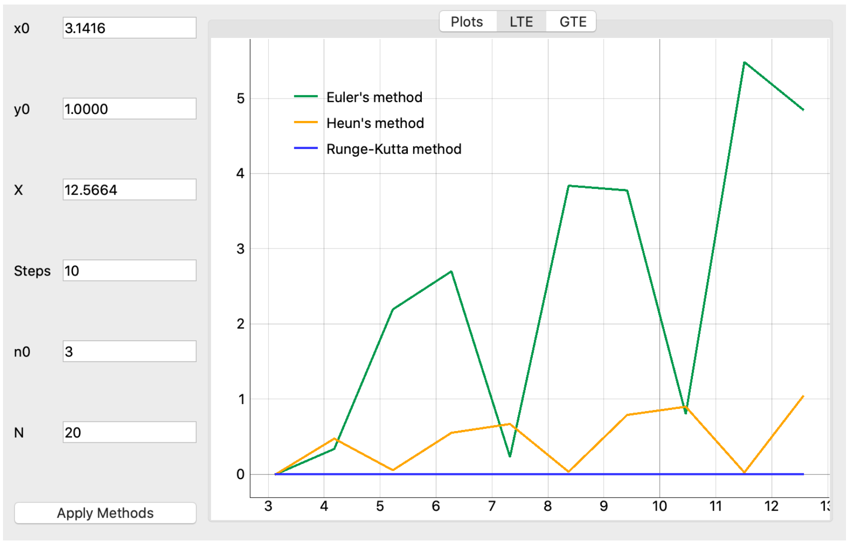
Using the mouse, it is possible to scale the plot like this


x0 3.1416
y0 1.0000
X 12.5664
Steps 10
n0 3
N 20

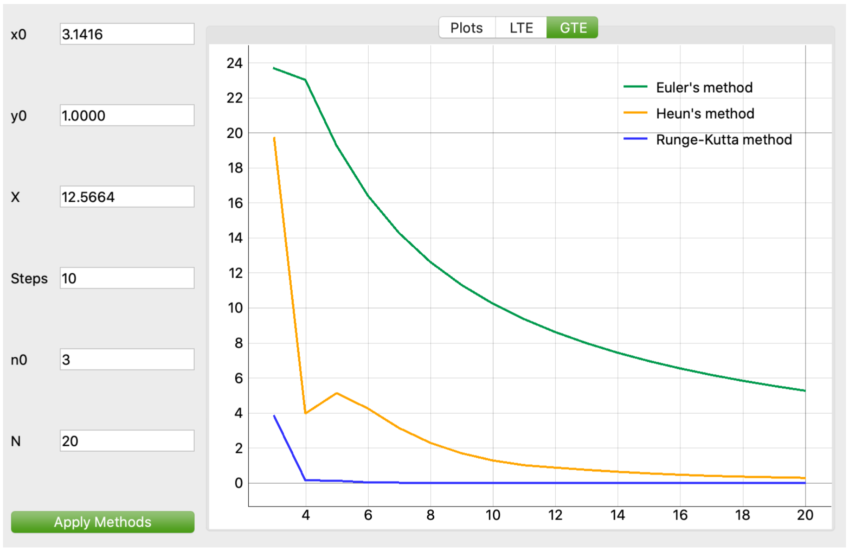
Using this button, it is possible to undo scaling



Further, we have two other plots: "LTE" (LTE at each 'x' point), "GTE" (maximal GTE depending on the number of steps)

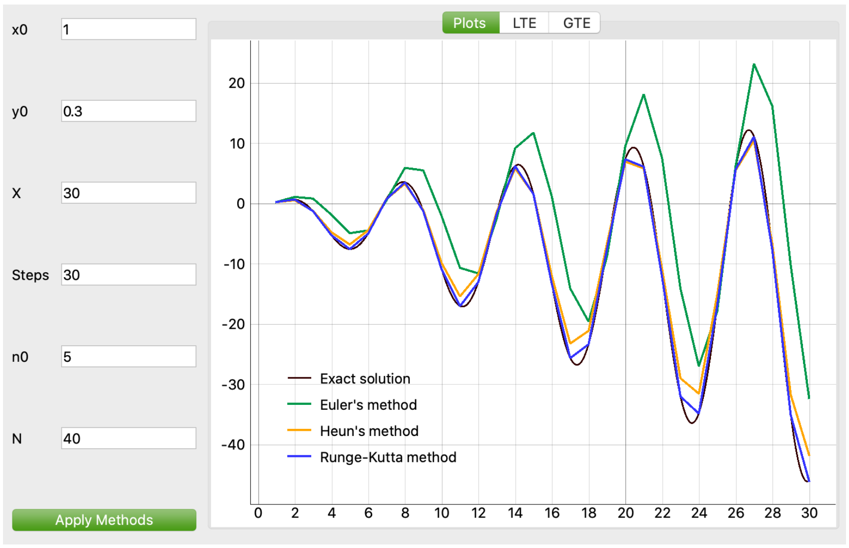


x0 3.1416
y0 1.0000
X 12.5664
Steps 10
n0 3
N 20

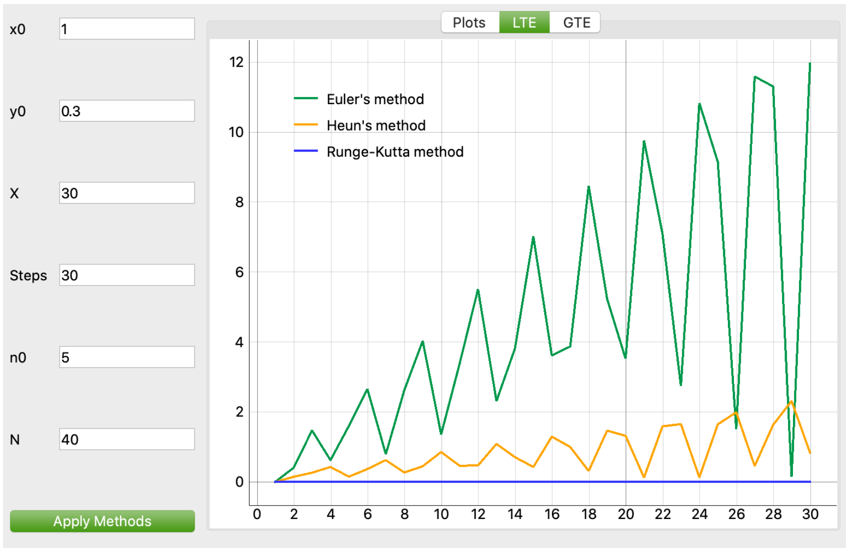


x0 3.1416
y0 1.0000
X 12.5664
Steps 10
n0 3
N 20

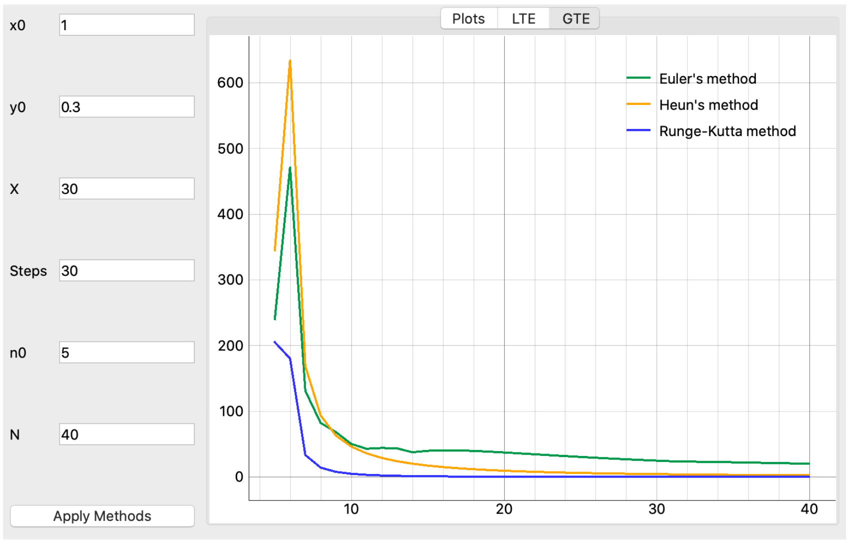
Here one may see results for some other initial values



x0: 1
y0: 0.3
X: 30
Steps: 30
n0: 5
N: 40



x0: 1
y0: 0.3
X: 30
Steps: 30
n0: 5
N: 40



x0: 1
y0: 0.3
X: 30
Steps: 30
n0: 5
N: 40

As we see, Runge-Kutta method provides us the best precision compared to Euler's and improved Euler's method.

Also, based on the "GTE" plots, we can conclude that Runge-Kutta method converges much faster.