Differential Equations

Computational practicum

Shabalin Dmitrii BS19-02

Variant 7

Plan:

Objectives:

- Learn to implement the exact solution of an IVP in our application
- Learn to implement Euler's, Improved Euler's and Runge-Kutta methods in our software application.
- Learn to provide data visualization capability (charts plotting) in the user interface of our application
- Learn to work with a GUI
- Investigate the convergence of these methods on different grid sizes
- Compare approximation errors of these methods plotting the corresponding chart for different grid sizes

Part I

1. Analytical Solution

***Problem*:** Solve IVP:

$$\begin{cases} y' = \frac{1}{x} + \frac{2y}{x \ln x} \\ y(2) = 0 \end{cases}$$

$x \in [2;\ 12]$

Solution:

This is First Order Linear Nonhomogeneous Differential Equation. Here we deal with Bernoulli equation.

Assume $\boldsymbol{x > 0}$ & $\boldsymbol{x \neq 1}$

Let's solve complementary equation:

$$y_c' - \frac{2y_c}{x \ln x} = 0$$

$$\int \frac{dy_c}{y_c} = \int \frac{2\,dx}{x \ln x}$$

$$\int \frac{dy_c}{y_c} = \int \frac{2\,dx}{x \ln x} = \bigg/ \begin{matrix} u = \ln x \\ dx = xdu \end{matrix} \bigg/ = \int \frac{2\,du}{u}$$

$$\ln|y_c| = 2\ln|\ln x| + C_1$$
$$y_c = C_2(\ln x)^2$$

Let us show that $y_c = (\ln x)^2$ is a partial solution of the complementary equation:

$$\left((\ln x)^2\right)' - \frac{2(\ln x)^2}{x \ln x} = 0$$

$$\frac{2(\ln x)}{x} - \frac{2(\ln x)}{x} = 0$$

$$0 = 0$$

To solve initial equation, we make a substitution $y = uy_c$. Substituting back into initial equation we get:

$$u'y_c + uy_c' - \frac{2uy_c}{x\ln x} = \frac{1}{x}$$

$$u'y_c + u\left(y_c' - \frac{2y_c}{x\ln x}\right) = \frac{1}{x}$$

Because y_c is complementary equation:

$$\left(y_c' - \frac{2y_c}{x\ln x}\right) = 0$$

So, we have:

$$u'y_c = f(x), \text{ where } f(x) = \frac{1}{x}$$

$$u' = \frac{f(x)}{y_c} = \frac{1}{x(\ln x)^2}$$

$$\int du = \int \frac{dx}{x(\ln x)^2} = \bigg/ \begin{matrix} v = \ln x \\ dx = xdv \end{matrix} \bigg/ = \int \frac{dv}{v^2}$$

$$u = -\frac{1}{\ln x} + C$$

$$y = uy_c = \left(-\frac{1}{\ln x} + C\right)(\ln x)^2 = C(\ln x)^2 - \ln x = \ln x\,(C\ln x - 1)$$

$$y = \ln x\,(C\ln x - 1)$$

Let's solve IVP:

$$\begin{cases} x_0 = 2 \\ y_0 = 0 \end{cases}$$

$$0 = \ln 2\,(C\ln 2 - 1)$$

$$C\ln 2 - 1 = 0$$

$$C = \frac{1}{\ln 2}$$

$$y = \ln x\left(\frac{\ln x}{\ln 2} - 1\right) = \frac{\ln x \ln\frac{x}{2}}{\ln 2}$$

Answer:

Our exact solution is

$$y = \frac{\ln x \ln\frac{x}{2}}{\ln 2}, \qquad where\ x > 0\ and\ x \neq 1$$

2. Solution for application

For application, we need to change the initial values, so we need to express C in terms of x_0 and y_0.

$$y = \ln x\,(C\ln x - 1)$$

$$y_0 = \ln x_0\,(C\ln x_0 - 1)$$

$$C = \frac{y_0 + \ln x_0}{\ln^2 x_0}$$

Also, our $f(x, y)$ are not continuous on $x \in (0;\ +\infty)$, because we have point of discontinuity when $x = 1$ (because $\ln(1) = 0$). So, we should split our interval into 2 parts: $x \in (0;\ 1)\ or\ x \in (1;\ +\infty)$

So, for the application we have:

$$y = \ln x\left(\left(\frac{y_0+\ln x_0}{\ln^2 x_0}\right)\ln x - 1\right) \text{ on interval } x \in (0;\ +\infty) \text{ and } x \neq 1$$

Part II

1. View explanation

For the computational practicum I used *Python3*. For plots I used the *pyqtgraph* library, and for the GUI I used *PyQt5,* and it was created in *Qt Creator* application.

There are several boxes for entering values:

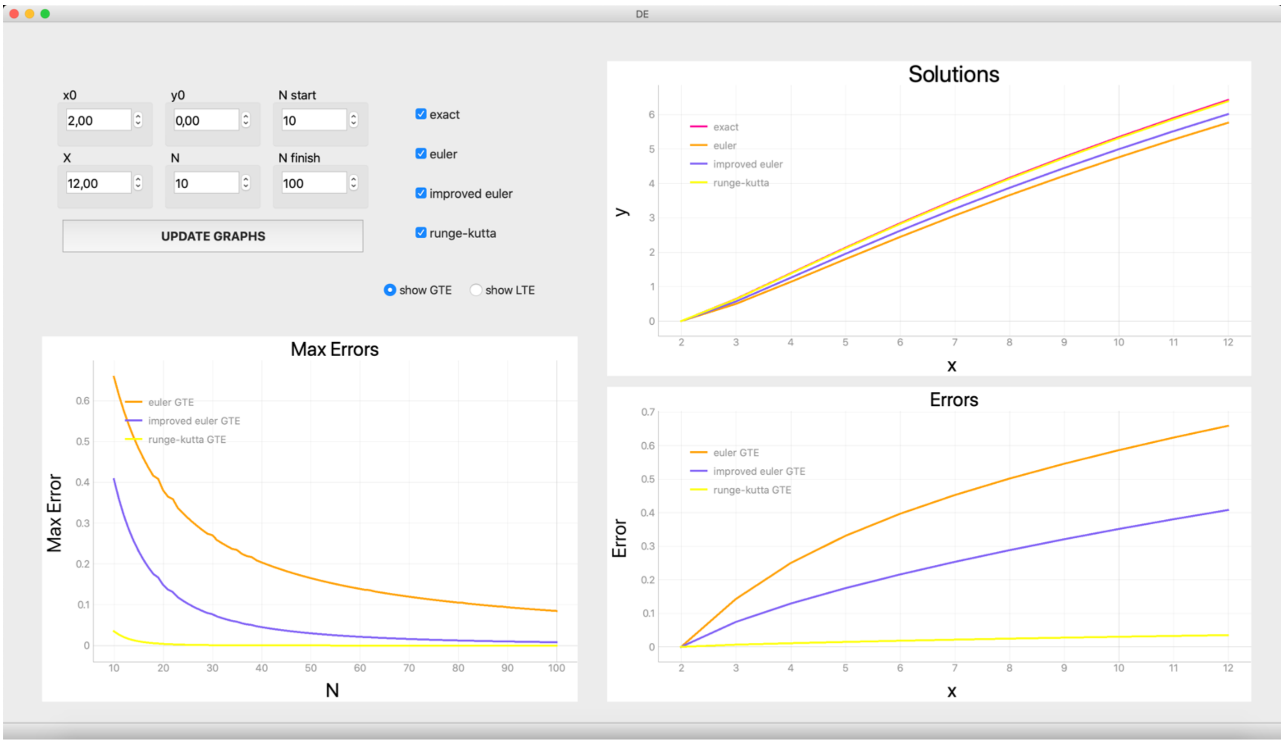
- x_0 - initial value of x
- y_0- initial value of y
- X - end point of the interval for x
- N - number of Grids
- N_{start} - start point of the interval for N
- N_{finish} - end point of the interval for N

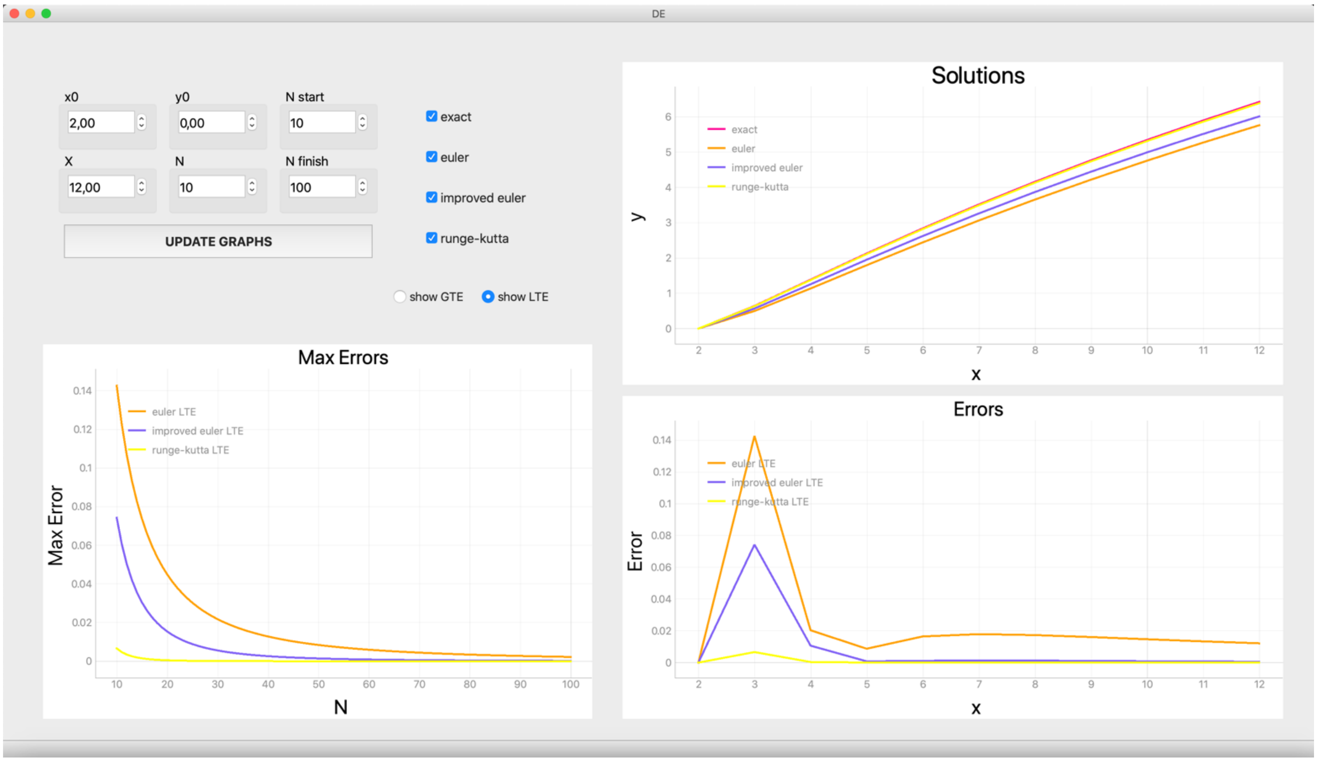
By default, they have values from our pdf task.

I have used several checkboxes to allow the user to choose which curves he/she wants to display in graph *Solutions*. They have default values, and if the user wants to change them, he must select the desired checkbox with the curve name. There are also radio buttons for plotting the error graph with two choices: *GTE* or *LTE*, which determine which graph will be displayed.

The GUI shows three graphs:

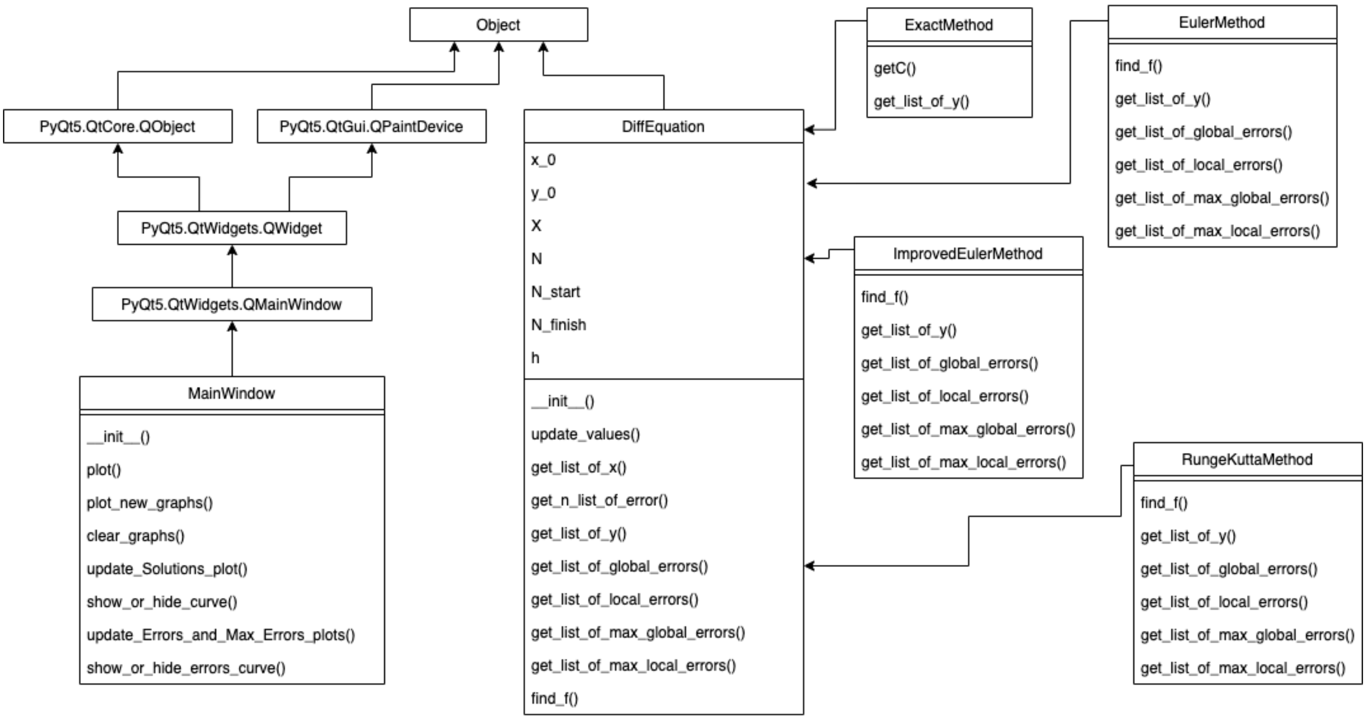
- *Solutions* - to display exact solution and also solutions after applying Euler, Improved Euler and Runge - Kutta methods
- *Errors* = to display LTE and GTE after applying Euler, Improved Euler and Runge - Kutta methods
- *Max Errors* = to display maximum LTE and maximum GTE after applying Euler, Improved Euler and Runge - Kutta methods on interval $[N_{start}\,;\ N_{finish}]$





2. Code explanation

https://github.com/shabalin13/DE_Assignment



There are six classes in the project. First of all, I have class *Solution* which has default methods for subclasses and methods that all subclasses will override.

```python
class DiffEquation:

    def __init__(self, window):
        self.x_0 = window.x_0.value()
        self.y_0 = window.y_0.value()
        self.X = window.X.value()
        self.N = int(window.N.value())
        self.h = (self.X - self.x_0) / self.N
        self.N_start = int(window.N_start.value())
        self.N_finish = int(window.N_finish.value())

    def update_values(self, x_0=None, y_0=None, X=None, N=None):...

    def get_list_of_x(self):
        return np.arange(self.x_0, self.X + self.h/2, self.h)

    def get_n_list_of_error(self):
        return np.arange(self.N_start, self.N_finish + 1, 1)

    def get_list_of_y(self, list_of_x):
        pass

    def get_list_of_global_errors(self, window):
        pass

    def get_list_of_local_errors(self, window):
        pass

    def get_list_of_max_global_errors(self, window):
        pass

    def get_list_of_max_local_errors(self, window):
        pass

    def find_f(self, x, y):
        pass
```

ExactMethod class uses the answer from the first part of the report with the ability to change x_0, y_0, X. The constant C is calculated automatically for given x_0, y_0.

```python
class ExactMethod(DiffEquation):

    def get_C(self):
        return (self.y_0 + math.log(self.x_0, math.e)) / math.pow(math.log(self.x_0, math.e), 2)

    def get_list_of_y(self, list_of_x):
        return [math.log(x, math.e) * (self.get_C() * math.log(x, math.e) - 1) for x in list_of_x]
```

EulerMethod class uses formula:

$$y_i = y_{i-1} + h \cdot f(x_{i-1}, y_{i-1}) \text{ , where } 1 \le i \le n$$

ImprovedEulerMethod class uses formula:

$$\begin{cases} K_1 = f(x_{i-1}, y_{i-1}) \\ K_2 = f(x_{i-1} + h, y_{i-1} + h \cdot K_1) \\ y_i = y_{i-1} + \frac{h}{2} \cdot (K_1 + K_2) \end{cases} \text{ , where } 1 \le i \le n$$

RungeKuttaMethod class uses formula:

$$\begin{cases} K_1 = f(x_{i-1}, y_{i-1}) \\ K_2 = f\left(x_{i-1} + \frac{h}{2}, y_{i-1} + \frac{h}{2} \cdot K_1\right) \\ K_3 = f\left(x_{i-1} + \frac{h}{2}, y_{i-1} + \frac{h}{2} \cdot K_2\right) \\ K_4 = f(x_{i-1} + h, y_{i-1} + h \cdot K_3) \\ y_i = y_{i-1} + \frac{h}{6} \cdot (K_1 + 2 \cdot K_2 + 2 \cdot K_3 + K_4) \end{cases} \text{ , where } 1 \le i \le n$$

All methods classes have methods, that responds for counting *global* and *local truncation errors* as an absolute value. GTE_i are calculated as an absolute value of difference between the exact solution and the approximated solution on i^{th} step. And LTE_{i+1} are also calculated as an absolute value of difference between the exact solution on $i + 1^{th}$ step and approximated solution on $i + 1^{th}$ step when there is no error in i^{th} step.

```python
    def get_list_of_global_errors(self, window):
        exact = ExactMethod(window)
        improved_euler = ImprovedEulerMethod(window)
        exact_list_of_y = exact.get_list_of_y(exact.get_list_of_x())
        improved_euler_list_of_y = improved_euler.get_list_of_y(exact.get_list_of_x())
        error_list = []
        for i in range(0, len(exact.get_list_of_x())):
            error_list.append(abs(exact_list_of_y[i] - improved_euler_list_of_y[i]))
        return error_list

    def get_list_of_local_errors(self, window):
        exact = ExactMethod(window)
        exact_list_of_y = exact.get_list_of_y(exact.get_list_of_x())
        error_list = [0.0]
        for i in range(0, len(exact.get_list_of_x()) - 1):
            x_i = self.x_0 + self.h * i
            error_list.append(abs(exact_list_of_y[i + 1] - (exact_list_of_y[i] + self.h * (
                        self.find_f(x_i, exact_list_of_y[i]) + self.find_f(x_i + self.h,
                                                                            exact_list_of_y[i] + self.h * self.find_f(
                                                                                x_i, exact_list_of_y[i])))/2)))

        return error_list
```

Also, all methods classes have methods, that responds for counting *max global* and *max local errors* on interval $[N_{start}\,;\ N_{finish}]$.

```python
    def get_list_of_max_global_errors(self, window):
        list_of_max_global_errors = []
        exact = ExactMethod(window)
        improved_euler = ImprovedEulerMethod(window)
        for j in self.get_n_list_of_error():
            exact.update_values(N=j)
            improved_euler.update_values(N=j)
            exact_list_of_y = exact.get_list_of_y(exact.get_list_of_x())
            improved_euler_list_of_y = improved_euler.get_list_of_y(exact.get_list_of_x())
            error_list = []
            for i in range(0, len(exact.get_list_of_x())):
                error_list.append(abs(exact_list_of_y[i] - improved_euler_list_of_y[i]))
            list_of_max_global_errors.append(max(error_list))
        return list_of_max_global_errors

    def get_list_of_max_local_errors(self, window):
        list_of_max_local_errors = []
        exact = ExactMethod(window)
        for j in self.get_n_list_of_error():
            exact.update_values(N=j)
            exact_list_of_y = exact.get_list_of_y(exact.get_list_of_x())
            error_list = [0.0]
            for i in range(0, len(exact.get_list_of_x()) - 1):
                x_i = exact.x_0 + exact.h * i
                error_list.append(abs(exact_list_of_y[i + 1] - (exact_list_of_y[i] + exact.h * (
                        self.find_f(x_i, exact_list_of_y[i]) + self.find_f(x_i + exact.h,
                                                                            exact_list_of_y[i] + exact.h * self.find_f(
                                                                                x_i, exact_list_of_y[i]))) / 2)))
            list_of_max_local_errors.append(max(error_list))
        return list_of_max_local_errors
```

If we consider *MainWindow* class we will see that *__init__* is responsible for assigning titles and labels to graphs, for updating graphs, as well as for connecting and using all buttons, checkboxes and radio buttons.

```python
def __init__(self, *args, **kwargs):
    super(MainWindow, self).__init__(*args, **kwargs)
    uic.loadUi('./form.ui', self)

    # connecting all buttons, checkboxes and radio buttons
    self.pushButton.clicked.connect(self.plot_new_graphs)

    self.exact_checkBox.stateChanged.connect(self.show_or_hide_curve)
    self.euler_checkBox.stateChanged.connect(self.show_or_hide_curve)
    self.improved_euler_checkBox.stateChanged.connect(self.show_or_hide_curve)
    self.runge_kutta_checkBox.stateChanged.connect(self.show_or_hide_curve)

    self.LTE_button.toggled.connect(self.show_or_hide_errors_curve)
    self.GTE_button.toggled.connect(self.show_or_hide_errors_curve)

    # assigning titles and labels to graphs
    self.Solutions.setTitle("Solutions", color='k', size='30pt')
    self.Solutions.setLabel('left', 'y', color='k', **{'font-size': '25pt'})
    self.Solutions.setLabel('bottom', 'x', color='k', **{'font-size': '25pt'})

    self.Errors.setTitle("Errors", color='000', size='25pt')
    self.Errors.setLabel('left', 'Error', color='000', **{'font-size': '25pt'})
    self.Errors.setLabel('bottom', 'x', color='000', **{'font-size': '25pt'})

    self.Max_Errors.setTitle("Max Errors", color='000', size='25pt')
    self.Max_Errors.setLabel('left', 'Max Error', color='000', **{'font-size': '25pt'})
    self.Max_Errors.setLabel('bottom', 'N', color='000', **{'font-size': '25pt'})

    self.plot_new_graphs()
```

Method *plot* sets the parameters of the graphs and plots them.

```python
# to set the parameters of the graphs and curves and plot them
def plot(self, graph, hour=None, temperature=None, color=None, name=None):
    if hour is None or temperature is None:
        hour, temperature, color = [], [], (1, 0, 0)

    graph.setBackground('w')
    graph.showGrid(x=True, y=True)

    graph.plot(hour, temperature, pen=pg.mkPen(color=color, width=5), name=name)
```

Method *plot_new_graphs* clear old graphs and builds new updated graphs. By default, the curves of all solutions are shown, as well as the GTE and max GTE.

```python
# plot solution, GTE and Max GTE curves for RungeKuttaMethod
runge_kutta_solution = RungeKuttaMethod(self)
self.plot(self.Solutions, runge_kutta_solution.get_list_of_x(),
          runge_kutta_solution.get_list_of_y(runge_kutta_solution.get_list_of_x()), color=(255, 255, 86),
          name="runge-kutta")
self.plot(self.Errors, runge_kutta_solution.get_list_of_x(),
          runge_kutta_solution.get_list_of_global_errors(self), color=(255, 255, 86),
          name="runge-kutta GTE")
self.plot(self.Max_Errors, runge_kutta_solution.get_n_list_of_error(),
          runge_kutta_solution.get_list_of_max_global_errors(self), color=(255, 255, 86),
          name="runge-kutta GTE")

# set checkboxes and radio button in default state
self.GTE_button.setChecked(True)
self.LTE_button.setChecked(False)
self.exact_checkBox.setChecked(True)
self.euler_checkBox.setChecked(True)
self.improved_euler_checkBox.setChecked(True)
self.runge_kutta_checkBox.setChecked(True)
```

(e.g. for runge-kutta method)

Method *clear_graphs* clear all graphs.

```python
def clear_graphs(self):
    graphs = [self.Solutions, self.Errors, self.Max_Errors]
    for graph in graphs:
        graph.clear()
    app.processEvents()
```

Method *show_or_hide_curve* is bound to checkboxes and is called if they have been changed. At the same time, it calls method *update_Solutions_plot* to update the *Solution* plot.

```
    # for checkboxes
    def show_or_hide_curve(self):
        self.update_Solutions_plot(self.exact_checkBox.isChecked(), self.euler_checkBox.isChecked(),
                                   self.improved_euler_checkBox.isChecked(),
                                   self.runge_kutta_checkBox.isChecked())
```

```
    # to plot Solution graph for new input values
    def update_Solutions_plot(self, isExact, isEuler, isImproved_euler, isRunge_kutta):
        self.Solutions.clear()
        if isExact:
            exact_solution = ExactMethod(self)
            self.plot(self.Solutions, exact_solution.get_list_of_x(),
                      exact_solution.get_list_of_y(exact_solution.get_list_of_x()), color=(255, 20, 147), name="exact")
        if isEuler:...
        if isImproved_euler:...
        if isRunge_kutta:...
        app.processEvents()
```

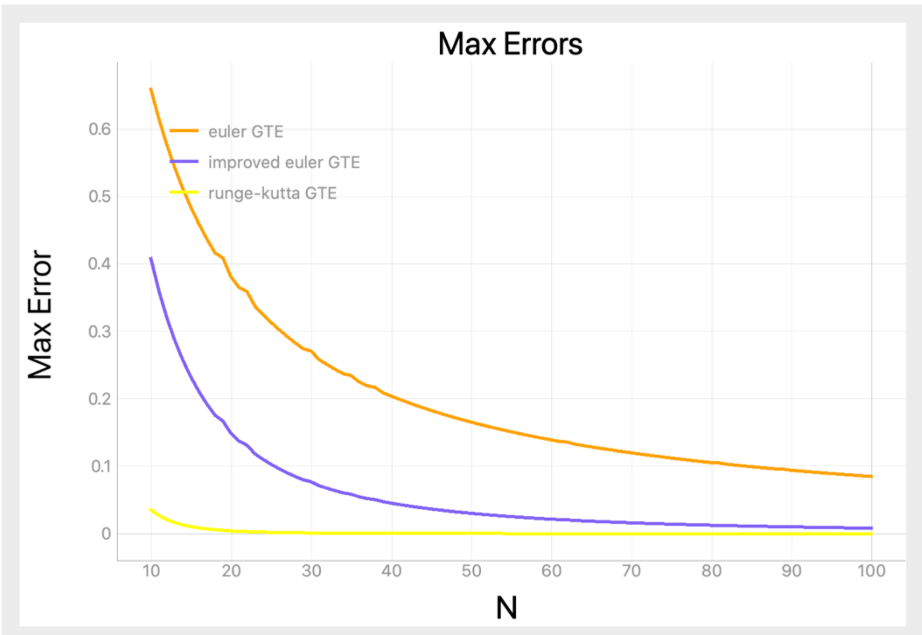
Also, method *show_or_hide_errors_curve* is bound to radio buttons and is called if they have been changed. At the same time, it calls method *update_Errors_and_Max_Errors_plots* to update the corresponding graphs.

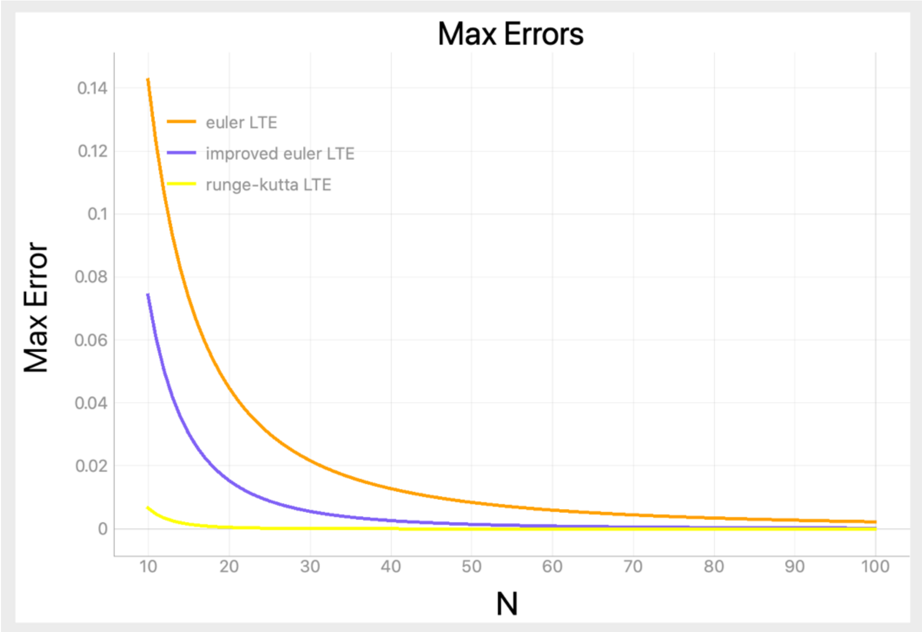
```
    # to plot new graphs after choosing GTE or LTE curves
    def update_Errors_and_Max_Errors_plots(self, isShowGTE):
        self.Errors.clear()
        self.Max_Errors.clear()
        # if we choose GTE graphs
        if isShowGTE:...
        # if we choose LTE graphs
        else:...
        app.processEvents()

    # for radio buttons
    def show_or_hide_errors_curve(self):
        self.update_Errors_and_Max_Errors_plots(self.GTE_button.isChecked())
```

Part III

1. Convergence analysis

The graphs show that the errors for the *Runge-Kutta method* are much smaller than the errors for the *Improved Euler method*, and for the *Improved Euler method*, the errors are much smaller than for the *Euler method*. And the more N, the less errors for the methods.





Conclusion:

I have created a software application using *GUI* in *Python*. I have implemented the *exact* solution of an IVP, as well as *Euler's*, *Improved Euler's* and *Runge-Kutta* methods. I built graphs of all solutions, as well as the *local* and *global errors*. I *analyzed* the *convergence* of these methods on different grid sizes, and *compared* the approximation *errors* of these methods.