Differential Equations Computational Practicum

Andrey Palaev

October 2020

1 Analytical solution

We need to solve the following equation:
$y' = 2e^x - y$, $y(0) = 0$
We can rewrite this equation in the form of Bernoulli's equation:
$y' + y = 2e^x$
Let's try a substitution $y = uy_c$,where y_c is the non-trivial solution of the complementary equation $y_c' + y_c = 0$. $y' = u'y_c + uy_c'$.
Let's solve the complementary equation. Since $y_c \neq 0$, we can divide by y_c:
$\frac{dy_c}{y_c} = -dx$
Integrating the equation, we get:
$\int \frac{dy_c}{y_c} = -\int dx$
$\ln|y_c| = -x$ (since we need any non-trivial solution, we can omit the constant)
$y_c = e^{-x}$
Let's substitute $y = uy_c$ and $y_c = e^{-x}$ in our equation:
$u'y_c + u(y_c' + y_c) = 2e^x$
$e^{-x}u' = 2e^x$ Let's solve obtained equation:
Since $e^{-x} \neq 0$, we can divide by e^{-x}:
$du = 2e^{2x}dx$
Integrating the equation, we get:
$\int du = \int 2e^{2x}dx$
$u = e^{2x} + C$, where $C \in \mathbb{R}$
Substituting u and y_c, we get:
$y = e^x + Ce^{-x}$
Let's substitute $y = 0$ and $x = 0$:
$0 = C + 1$
Hence, $C = -1$ and $y = e^x - e^{-x}$
Analysis of points of discontinuity:
The initial equation does not contain constraints on x and y as well as the solution. We can express the constant C from the most general solution as $C = \frac{y-e^x}{e^{-x}}$, and it also has no constraints neither on C nor on x.

2 Implementation

For the assignment, I decided to use Python language with the following imported libraries:

1. numpy - for arrays (numpy.array is easier to use and more powerful then usual lists)
2. matplotlib - for plotting graphs
3. PyQT5 - for Graphical User Interface
4. copy - for copying objects
5. sys - for interacting with the interpreter

You can see the code going to the GitHub repo: `https://github.com/Palandr1234/DE_Assignment`
I implemented the class **Equation** which contains just parameters of our differential equation: x_0, y_0, X, n, f, where f is the function in $y' = f(x, y)$ and the solution of the equation. f and the solution was provided as lambda functions.
I created the parent class **Method** for all methods (both numerical and analytical) which contains the following methods:

1. solve() for the applying the method that is different for all child classes
2. LTE() and GTE() for computing the LTE and the changes of the maximum of GTE respectively which are the same for all methods

Here is the code of the **Method** class:

```
class Method:
    # the function for deriving h, x in all steps and y0
    def solve(self, eq):
        h = (eq.X - eq.x0)/(eq.n - 1)
        x = np.linspace(eq.x0, eq.X, eq.n)
        y = np.zeros(x.shape[0])
        y[0] = eq.y0
        return h, x, y

    # the function for computing the LTE for the method
    def LTE(self, x, eq):
        LTE = np.zeros(x.shape[0])
        # compute the exact solution
        _, y_exact, _ = Exact().solve(eq)
        for i in range(1, x.shape[0]):
            # compute new y_appr by this method using x and y_exact from the previous step
            eq_appr = Equation(x[i-1], y_exact[i-1], x[i], 2, eq.solution, eq.f)
            _, appr, _ = self.solve(eq_appr)
            # compute the LTE
            LTE[i] = abs(y_exact[i] - appr[-1])
        return LTE

    # compute the maximum of the GTE for every n from no to N
    def GTE(self, eq, n0, N):
        # copy the given equation
        eq_appr = copy.deepcopy(eq)
        ans = np.zeros(N-n0+1)
        for i in range(n0, N+1):
            # change the number of steps of the equation
            eq_appr.n = i
            # get the solution by the method
            _, y, _ = self.solve(eq_appr)
            # get the exact solution
            _, y_exact, _ = Exact().solve(eq_appr)
            # compute the maximum GTE
            ans[i-n0] = max(abs(y - y_exact))
        return ans
```

I created the classes **Exact** (Exact solution), **Euler** (Euler's method), **ImprovedEuler** (Improved Euler's method) and **RungeKUtta** (Runge-Kutta method).
Also, I have classes for Main Window (**MyWindow**) and for Tab Widget (**MyTableWidget**). The first one contains __init__() function which initializes all labels, lineEdits, layouts, widgets etc. The second one contains the __init__() for the initialization of tabs and plot() for replotting the graphs.

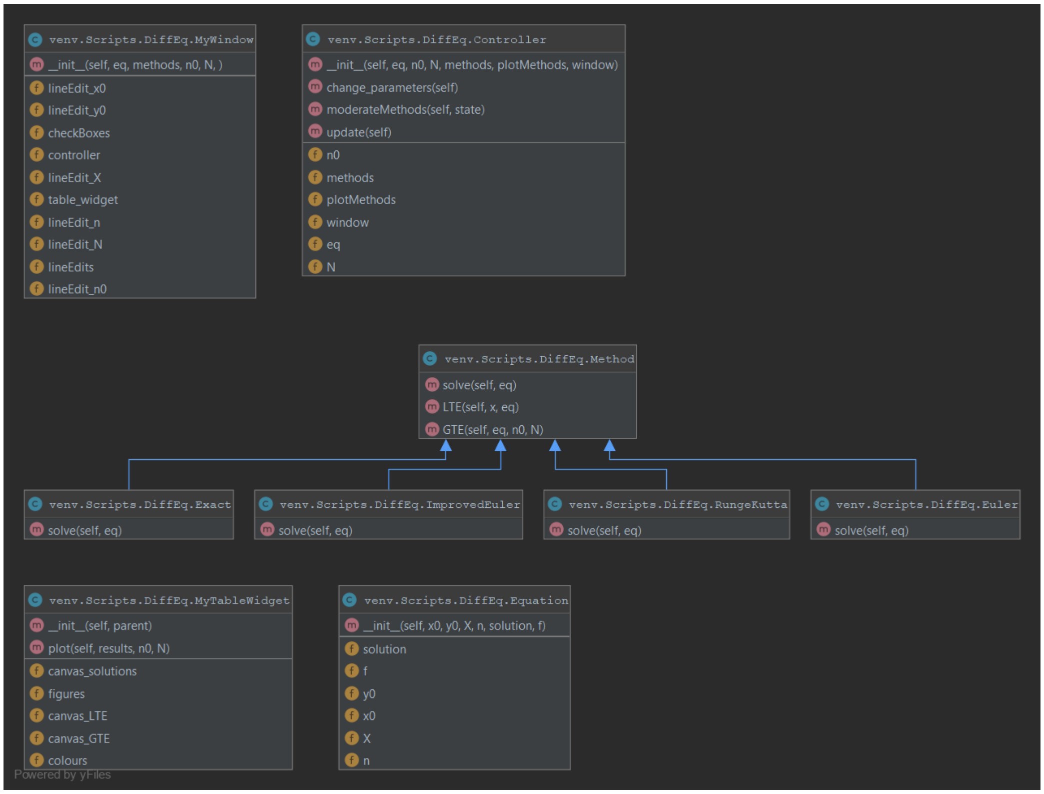
I also implemented the class **Controller** for the reading the input, updating the parameters and providing the updated results (using the Model) to the View. In addition to __init__() which just initializes all the parameters, this class has the change_parametrs() function for reading the parameters when the user presses the button "Plot", moderateMethods() for checking which methods needs to be added to the graphs and which needs to be removed from the graphs when the user changes the state of one of the checkboxes for the methods and update() for updating the results of computations.
As a results, my code can be splitted into three parts: View, Model and Controller.

Here is the code for moderateMethods() and update() functions of the **Controller** class:

```python
# the method for adding the methods to the graphs / removing it
def moderateMethods(self, state):
    # if the user wants to add some method to the graphs
    if state == QtCore.Qt.Checked:
        # find the method the user wants to add
        for i in range(len(self.window.checkBoxes)):
            if self.window.sender() == self.window.checkBoxes[i]:
                # set that we need to plot this method
                self.plotMethods[i] = True
    # if the user wants to remove some method from the graph
    else:
        # find the method the user wants to remove
        for i in range(len(self.window.checkBoxes)):
            if self.window.sender() == self.window.checkBoxes[i]:
                # set that we don't need to plot this graph
                self.plotMethods[i] = False
    # update the results
    self.update()
```

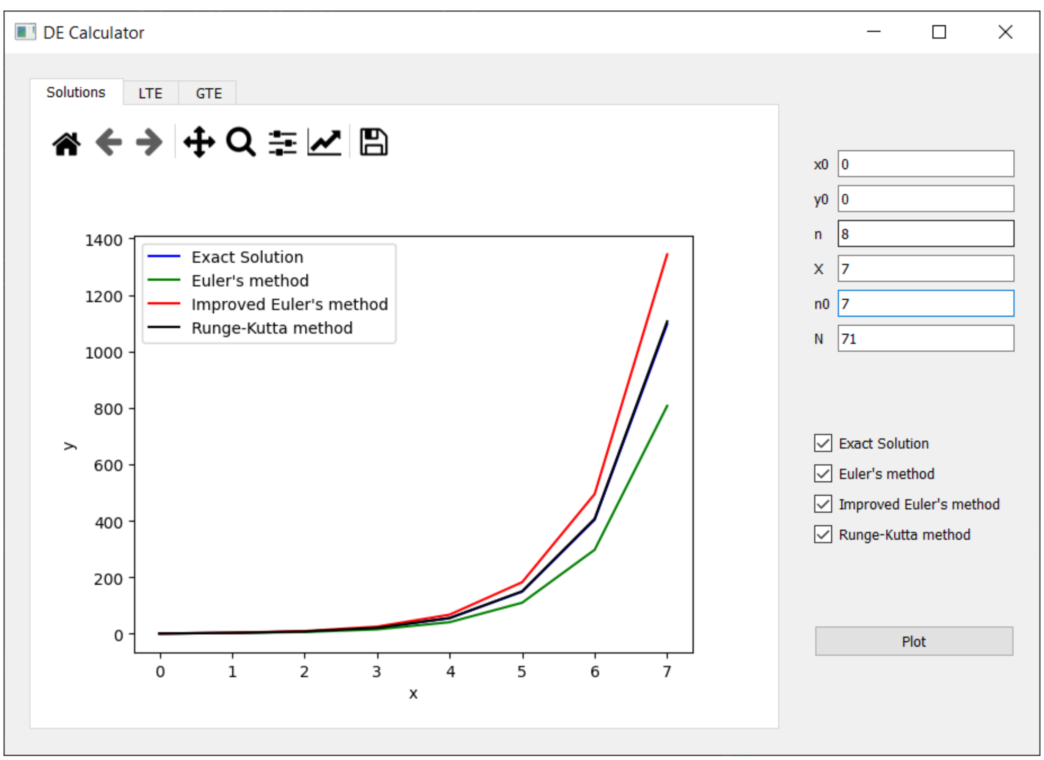
Here is the code for the plot() method of the **MyTableWidget** class:

```python
# function for replotting the graphs
def plot(self, results, n0, N):
    axes = []
    # clear all the figures and add new subplots
    for figure in self.figures:
        figure.clear()
        axes.append(figure.add_subplot(111))
    for (plotMethod, x, y, label, lte, gte), colour in zip(results, self.colours):
        # if we don't need to plot this method, just skip it
        if plotMethod is False:
            continue
        # plot the results of the method
        axes[0].plot(x, y, colour, label=label)
        axes[0].set_xlabel("x")
        axes[0].set_ylabel("y")
        axes[0].legend()
        # plot LTe of the second tab
        axes[1].plot(x, lte, colour, label=label)
        axes[1].legend()
        axes[1].set_xlabel("x")
        axes[1].set_ylabel("LTE")
        # plot the maximum of GTE on the third tab
        axes[2].plot(np.arange(n0, N + 1), gte, colour, label=label)
        axes[2].legend()
        axes[2].set_xlabel("n")
        axes[2].set_ylabel("The maximum of GTE")
    # show the graphs
    self.canvas_solutions.draw()
    self.canvas_LTE.draw()
    self.canvas_GTE.draw()
```

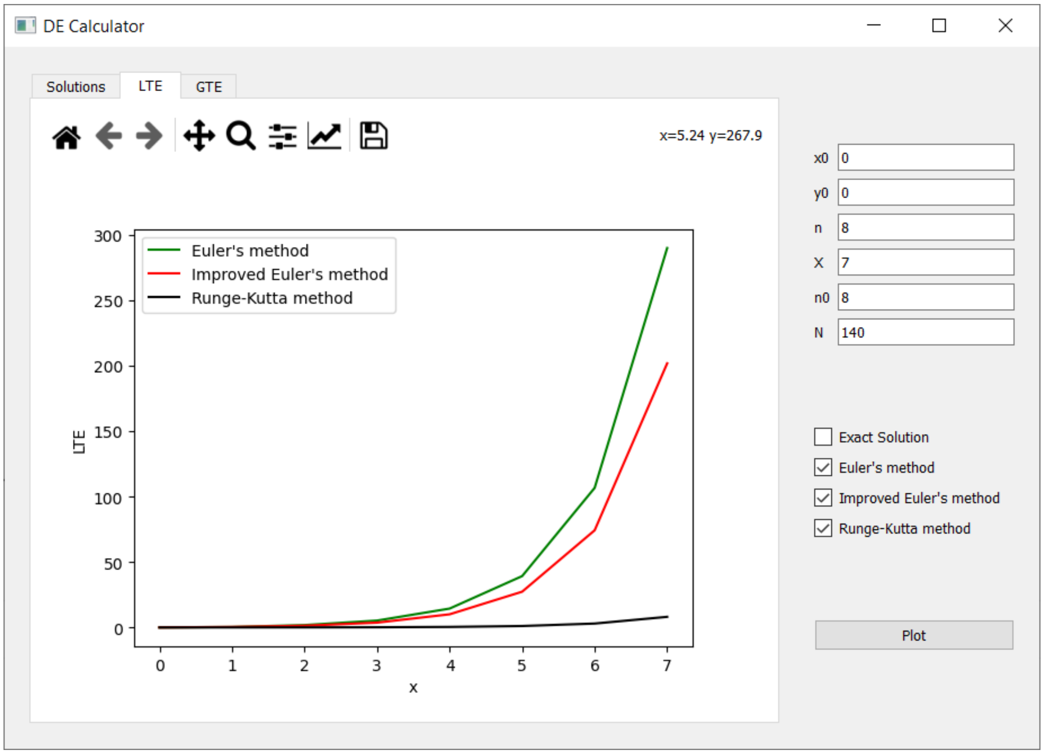


UML diagram

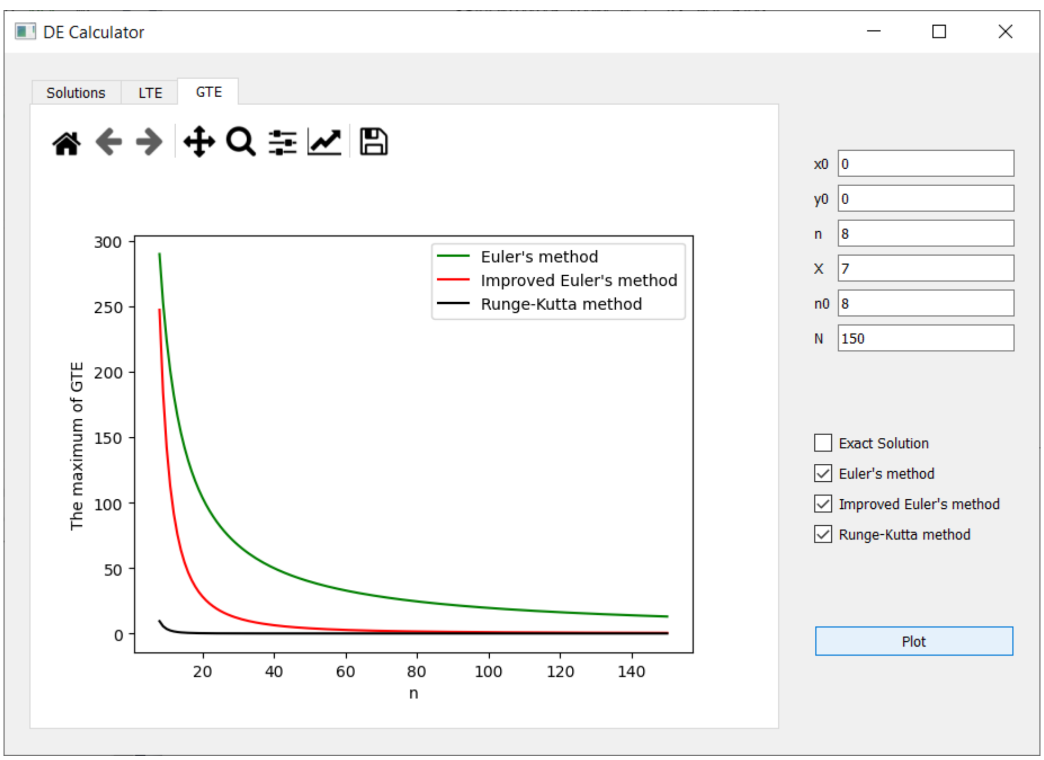
3 Plotting



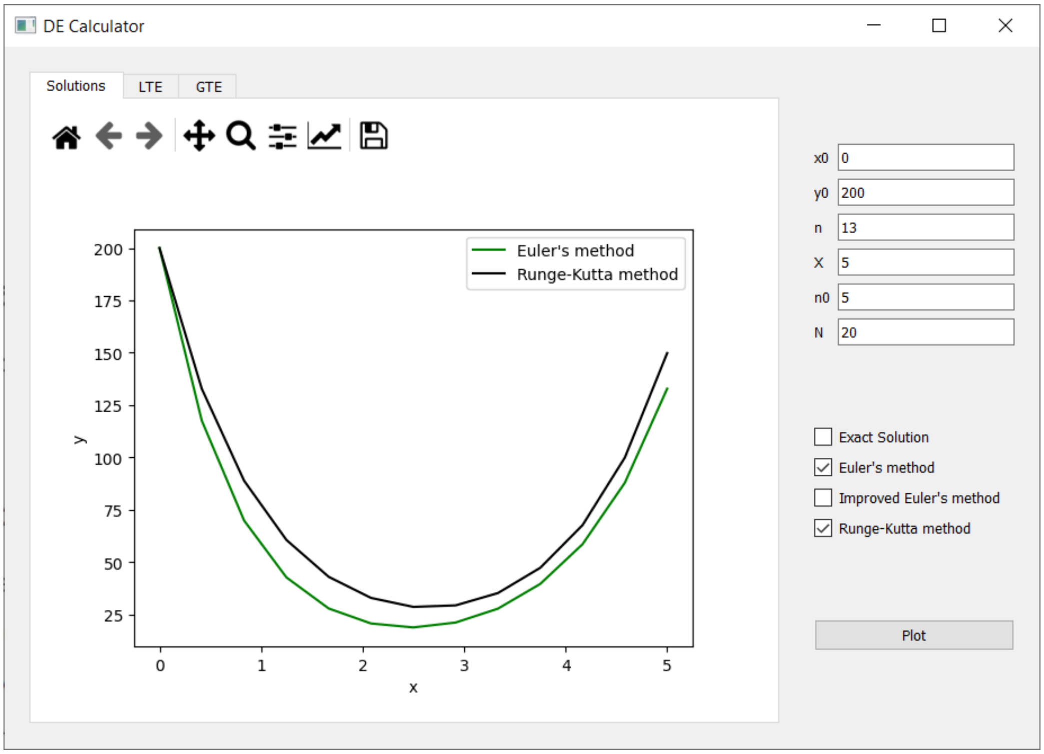
Results of all the methods for $x_0 = 0$, $y_0 = 0$, $X = 7$ and $n = 8$ (Exact solution and results of Runge-Kutta method coincide)



LTE for $x_0 = 0$, $y_0 = 0$, $X = 7$ and $n = 8$



the maximum of GTE as a function of n for $x_0 = 0$, $y_0 = 0$, $X = 7$, $n = 8$, $n_0 = 8$ and $N = 150$. For all methods the maximum of GTE decreases as n increases and the maximum of GTE tends to 0



In the GUI, the user can input x_0, y_0, n, X, n_0 and N and can set which methods should be plotted and which shouldn't