COMPUTATIONAL PRACTICUM

Lymarenko Lev (05.07.2001)
l.lymarenko@innopolis.university

October 18, 2020

1 Solving DE

$$y' = 3y^{\frac{2}{3}}$$

Divide by $y^{\frac{2}{3}}$, we can assume $y \neq 0$ because $y = 0$ is a trivial solution

$$\frac{y'}{y^{\frac{2}{3}}} = 3$$

Integrating both sides gives us:

$$3y^{\frac{1}{3}} = 3x + C$$
$$y = (x + C)^3$$

Express C for IVP:

$$C = y^{\frac{1}{3}} - x, C = y_0^{\frac{1}{3}} - x_0$$

Eventually:

$$y = (x + (y_0^{\frac{1}{3}} - x_0))^3 \text{ where } x_0, y_0 \text{ are some constants from IVP}$$

2 Code

First, project has **constants.py** file, where you can easily change differential equation (line 20), solution (line 23), some initial values (lines 11-17) and other settings of the project

```
3   APP_TITLE = 'ODE solver v1.0'
4   APP_SIZE = '1200×600' # in pixels
5   LOG_FORMAT = '%(levelname) -10s %(asctime)s %(name) -15s %(funcName) -10s: %(message)s'
6   UPDATE_BUTTON_NAME = 'Update'
7   RESIZE = True
8   LOG_LEVEL = logging.INFO
9
10  INITIAL_VALUES = [
11      2,      # x_0
12      1,      # y_0
13      10,     # x_n
14      50,     # number of iterations
15      10,     # n_start
16      100,    # n_end
17      'y\' = 3y^(2/3)', # title of eq
18  ]
19
20  f = lambda x,y: 3 * y**(2/3)
21  def y_solution_getter(x_0, y_0):
22      def solution(x):
23          return (x + (y_0**(1/3) - x_0))**3
24      return solution
```

In the project you can find some supporting classes, like **Function**, **DifferentialEq** which makes working with data more convenient and **Manager** that help to connect front and back end of app

An interesting point that **DifferentialEq** class has fields `solution` and `h` that are actually properties (something like a function without arguments). It means that there is no need to update `solution` and `h`, you just need to change x_0, x_n, y_0 or n

```python
@property
def solution(self):
    return self.f_solution(self.x_0, self.y_0)

@property
def h(self):
    return (self.x_n - self.x_0) / self.n
```

Since we need to update values of differential equation, I suggest not to mix front and back end, but create a bridge between them, that will listen front-end, do some work and update shared values. That is why I need **Manager** (**manager.py**). It has all the necessary attributes for managing and updating and 2 public functions: `update()` and `hide_methods()`. These are the two functions that the front-end will call.

```python
class Manager:
    '''
    Class for managing changes of values of diff. eq
    '''
    def __init__(self, diff: DifferentialEq, methods: List[Method], solution_color):
        self.diff = diff
        self.methods = methods
        self.functions: Dict[List[Function]] = {
            str(method):[Function(name=str(method), color=method.color) for _ in range(len(methods))]
                for method in methods
            }
        self.solution = Function(name='Analytical solution', color=solution_color)
        self.update()

    def update(self, x_0=None, y_0=None, x_n=None, n=None, n_start=None, n_end=None):…

    def __update_solution(self):…

    def hide_methods(self, methods: Dict[str, bool]):…
```

To implement 3 iteration methods, I decided to write an interface **Method** (**de_solver.py**). It has `get_next_value()` function, that we need to implement, and `solve()` function, that actually solves differential equation based on implemented function

```python
class Method:
    def __init__(self, color=None):…

    def get_next_value(self, f, x_i, y_i, h):
        raise NotImplementedError('you should implement it first')

    def solve(self, diff: DifferentialEq, precision=4) -> Tuple[Function]:…


    def __execute_method(self, f, y_solution_func, x_0, y_0, x_n, n, precision=4) -> Tuple[Function]:…
```

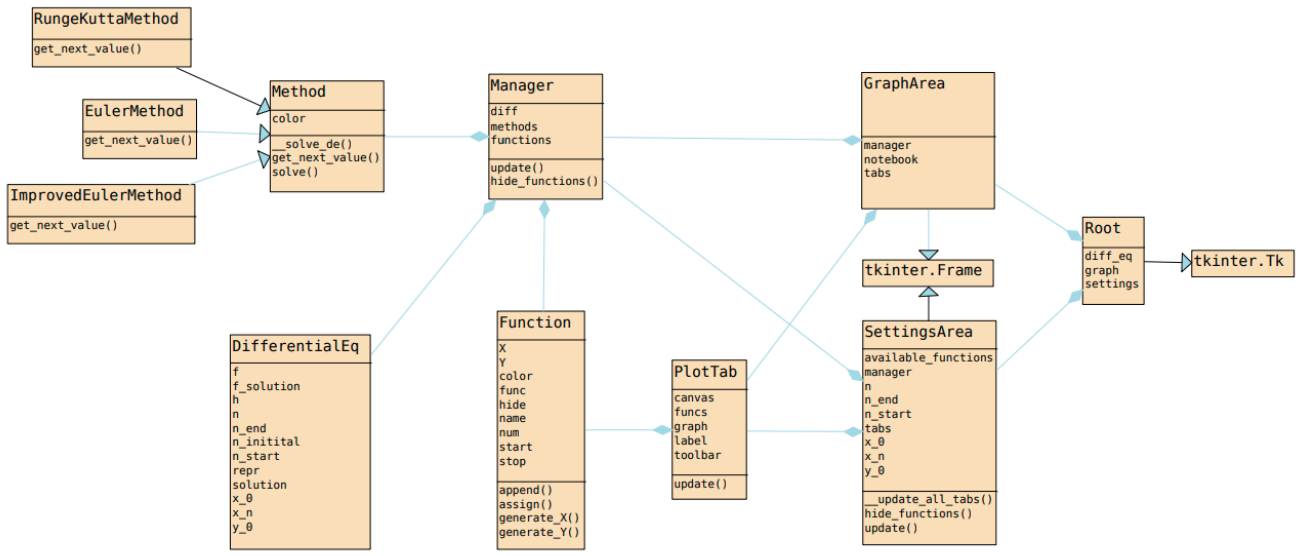
And then implement 3 derived classes
EulerMethod, **ImprovedEulerMethod** and **RungeKuttaMethod** (**de_solver.py**):

```python
class EulerMethod(Method):
    def get_next_value(self, f, x_i, y_i, h):
        # https://en.wikipedia.org/wiki/Euler_method
        return y_i + h * f(x_i, y_i)


class ImprovedEulerMethod(Method):
    def get_next_value(self, f, x_i, y_i, h):
        # https://en.wikipedia.org/wiki/Heun%27s_method
        k_1i = f(x_i, y_i)
        k_2i = f(x_i + h, y_i + h * f(x_i, y_i))
        y_i = y_i + h/2 * (k_1i + k_2i)
        return y_i

class RungeKuttaMethod(Method):
    def get_next_value(self, f, x_i, y_i, h):
        # https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods
        k_1i = f(x_i, y_i)
        k_2i = f(x_i + h/2, y_i + h/2*k_1i)
        k_3i = f(x_i + h/2, y_i + h/2*k_2i)
        k_4i = f(x_i + h, y_i + h*k_3i)

        return y_i + h/6*(k_1i+2*k_2i+2*k_3i+k_4i)
```

Also I splitted front-end into 2 parts: **SettingsArea** and **GraphArea** (**frames.py**). SettingsArea should update the Manager, GraphArea should update the plotting

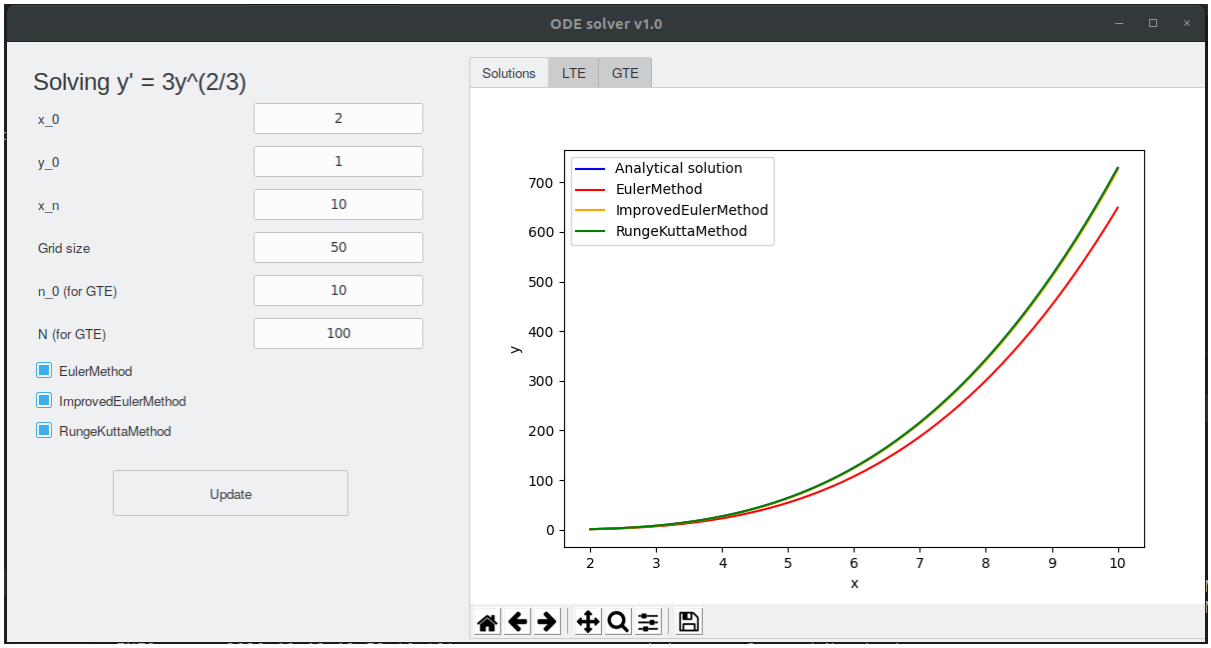
3 UML diagram

Here you can see UML diagram of my project structure

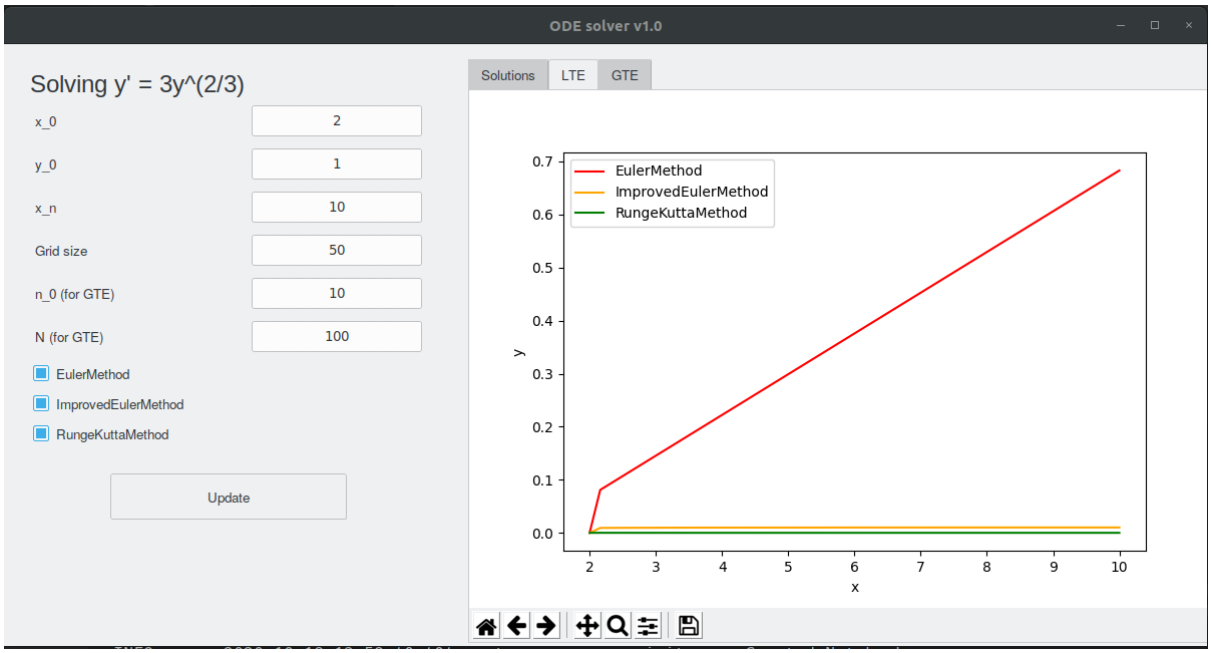


4 Screenshots of APP

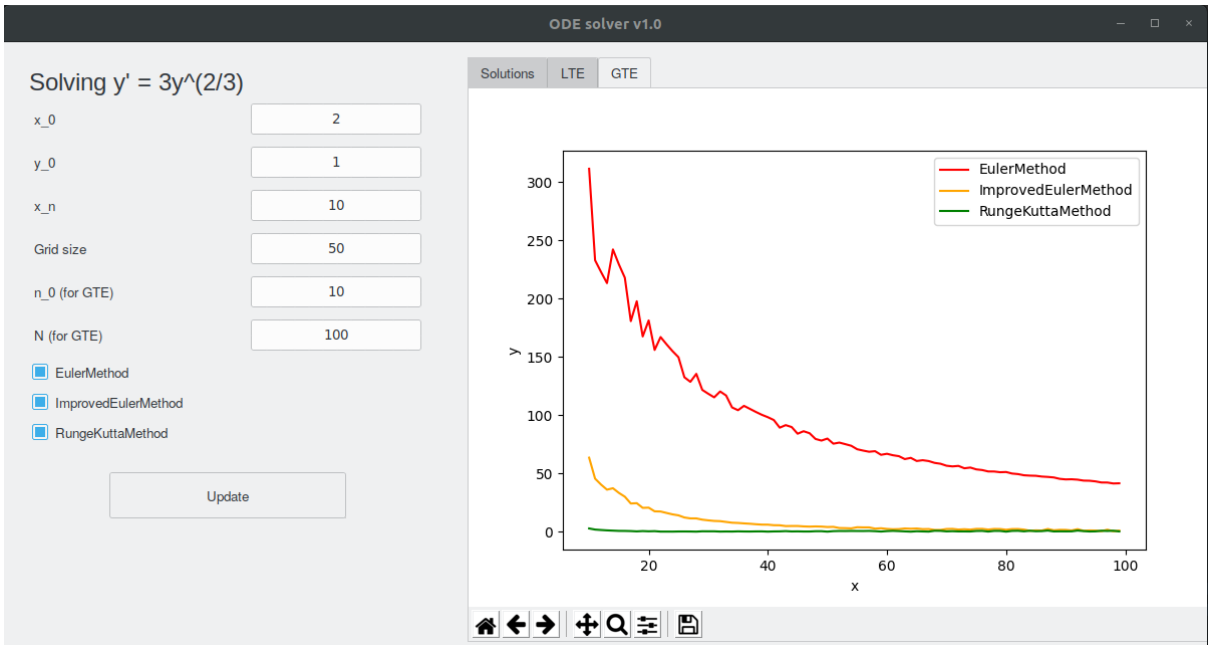
Solution tab



LTE tab



GTE tab



Example of chart analysis