DIFFERENTIAL EQUATIONS COMPUTATIONAL PRACTICUM

Rufina Talalaeva, BS18-01, variant 19

October 2019

1 Exact Solution

Solving the initial value problem with the ODE of first order.

$$\begin{cases} y' = 2x + y - 3; \\ y(1) = 1; \\ x \in (1, 7). \end{cases}$$

This equation is non-homogeneous(λ did not expire): $y' = 2\lambda x + \lambda y - 3$. We will solve by Bernoulli's method. So, let's make substitution $y = uv$, where $u = u(x)$ and $v = v(x)$, then $y' = u'v + uv'$. Our equation after substitution:

$u'v + uv' - uv = 2x - 3.$
$u'v + u(v' - v) = 2x - 3.$
$\begin{cases} v' - v = 0; \\ u'v = 2x - 3. \end{cases}$

1)From first equation we will find v function:
$v' - v = 0;$
$\frac{dv}{v} = dx;$
$\int \frac{dv}{v} = \int dx;$
$\ln|v| = x;$
$v = e^x$. Function v is found.

2)Let's substitute the function v from 1) into the second equation and find u function:
$u'v = 2x - 3;$
$u'e^x = 2x - 3;$
$\frac{du}{dx} = (2x - 3)e^{-x};$
$du = (2x - 3)e^{-x}dx;$
$\int du = \int (2x - 3)e^{-x}dx;$
$u = -(2x-3)e^{-x} - \int -2e^{-x} = -(2x-3)e^{-x} - 2e^{-x} + C = -2xe^{-x} + e^{-x} + C.$

3) General solution:
$y = uv = (-2xe^{-x} + e^{-x} + C)e^{x} = -2x + 1 + Ce^{x}$, where C - const.

4) Solving initial value problem to find constant:
$y = -2x + 1 + Ce^{x}$, where C - const.
$1 = -2 + 1 + Ce^{1}$;
$C = \frac{2}{e}$.
So, the solution for initial value problem is: $y = -2x + 1 + 2e^{x-1}$.

2 Implementation

Classes

There are 7 classes in my project: Application, Field, Solution, Exact, Euler, Improved Euler and Runge Kutta. If there is another function for solving IVP you need to change only 3 rows of code(solution.py: 49, 54, 103).
Application - class for providing GUI, an interaction between user input and computations. Also it is responsible for updating info from input fields and checkbox.
Field - class for textboxes that provide their initialization, drawing, and evaluation of data(their could be inserted expression, not just numbers)
Solution - abstract class for solutions, has the same function recalculate and calculate error for all methods.
Exact, Euler, Improved Euler, Runge Kutta - classes inherited from Solution. Each of them has it's own solve function, graph function and graph error(only for Exact there is no such function, because error obviously is 0.
At the figure 1 there is UML class diagram of the project.

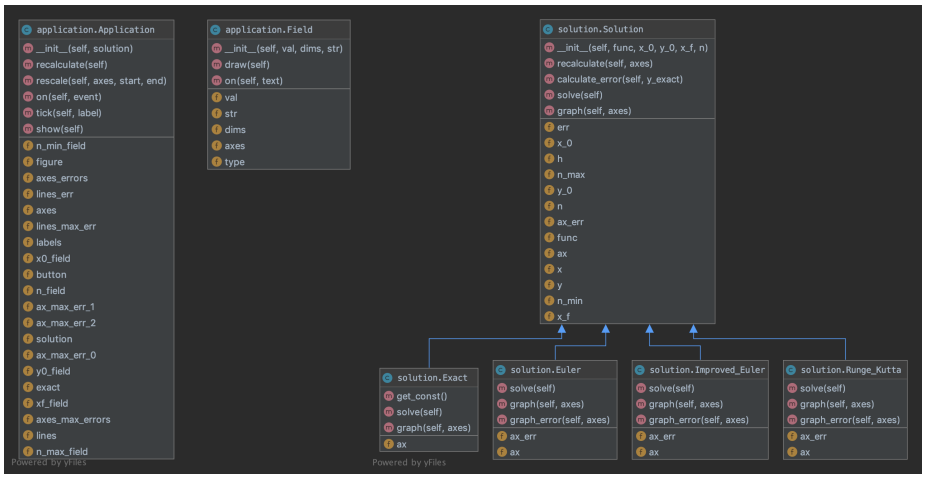


Figure 1: UML Class diagram

Implementation of exact solution and numerical methods

Initialization for each method:

```
11  class Solution(object):
12      def __init__(self, func, x_0, y_0, x_f, n):
13          self.x_0 = x_0
14          self.y_0 = y_0
15          self.x_f = x_f
16          self.n = n
17          self.n_min = n - 10
18          self.n_max = n + 10
19          self.func = func
20          self.h = (x_f - x_0) / (n)
21          self.x = np.linspace(x_0, x_f, n + 1)
22          self.y = np.zeros([n + 1])
23          self.err = np.zeros([n + 1])
24          self.y[0] = y_0
25          self.ax = matplotlib.lines.Line2D([],[])
26          self.ax_err = matplotlib.lines.Line2D([],[])
27          self.solve()
```

Solve Function for each method:

```
49      def get_const(self):
50          return (self.y_0 + 2 * self.x_0 - 1) / math.exp(self.x_0)
51
52      def solve(self):
53          const = self.get_const()
54          for i in range(1, self.n + 1):
55              self.y[i] = (-2) * self.x[i] + 1 + const * math.exp(self.x[i])
```

(a) Exact

```
61      def solve(self):
62          for i in range(1, self.n + 1):
63              self.y[i] = self.h * self.func(self.x[i - 1], self.y[i - 1]) + self.y[i - 1]
```

(b) Euler

```
72      def solve(self):
73          for i in range(1, self.n + 1):
74              self.y[i] = self.h * self.func(self.x[i - 1], self.y[i - 1]) + self.y[i - 1]
75              self.y[i] = (self.func(self.x[i - 1], self.y[i - 1]) + self.func(self.x[i], self.y[i]))
76              self.y[i] = self.h * self.y[i] / 2 + self.y[i - 1]
```

(c) Improved Euler

```
85      def solve(self):
86          k = np.zeros([4])
87          for i in range(1, self.n + 1):
88              k[0] = self.func(self.x[i - 1], self.y[i - 1])
89              k[1] = self.func(self.x[i - 1] + self.h / 2, self.y[i - 1] + self.h * k[0] / 2)
90              k[2] = self.func(self.x[i - 1] + self.h / 2, self.y[i - 1] + self.h * k[1] / 2)
91              k[3] = self.func(self.x[i - 1] + self.h, self.y[i - 1] + self.h * k[2])
92              self.y[i] = (self.h / 6) * (k[0] + 2 * k[1] + 2 * k[2] + k[3]) + self.y[i - 1]
```

(d) Runge Kutta

Implementation of errors

Local error was calculated as difference between exact and approximation value of y of numerical method. This function is the same for all methods, so it was implemented in Solution abstract class.

```
37          def calculate_error(self, y_exact):
38              self.err = y_exact - self.y
39              self.ax_err.set_data(self.x, self.err)
```

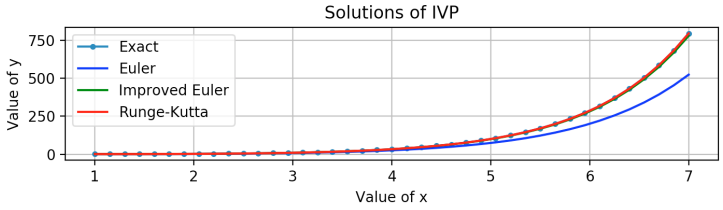
Total error was calculated as maximum absolute value of local error of numerical method. These calculations were implemented in Application class.

```
88          n_values = np.linspace(self.n_min_field.val, self.n_max_field.val, self.n_max_field.val - self.n_min_field.val + 1)
89          e_total_errors = np.zeros([self.n_max_field.val - self.n_min_field.val + 1])
90          i_total_errors = np.zeros([self.n_max_field.val - self.n_min_field.val + 1])
91          r_total_errors = np.zeros([self.n_max_field.val - self.n_min_field.val + 1])
92          for i in range(self.n_min_field.val, self.n_max_field.val + 1):
93              exact = sol.Exact(sol.function, self.x0_field.val, self.y0_field.val, self.xf_field.val, i)
94              euler = sol.Euler(sol.function, self.x0_field.val, self.y0_field.val, self.xf_field.val, i)
95              i_euler = sol.Improved_Euler(sol.function, self.x0_field.val, self.y0_field.val, self.xf_field.val, i)
96              r_k = sol.Runge_Kutta(sol.function, self.x0_field.val, self.y0_field.val, self.xf_field.val, i)
97              euler.calculate_error(exact.y)
98              i_euler.calculate_error(exact.y)
99              r_k.calculate_error(exact.y)
100             e_total_errors[i - self.n_min_field.val] = max(np.amax(euler.err), abs(np.amin(euler.err)))
101             i_total_errors[i - self.n_min_field.val] = max(np.amax(i_euler.err), abs(np.amin(i_euler.err)))
102             r_total_errors[i - self.n_min_field.val] = max(np.amax(r_k.err), abs(np.amin(r_k.err)))
103             del exact
104             del euler
105             del i_euler
106             del r_k
```

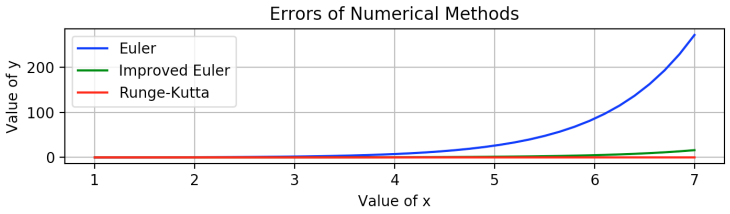
3 Interface

I used *matplotlib* library for making GUI: graphs, text fields, buttons and check-boxes, for fast calculations I prefered *numpy* library.

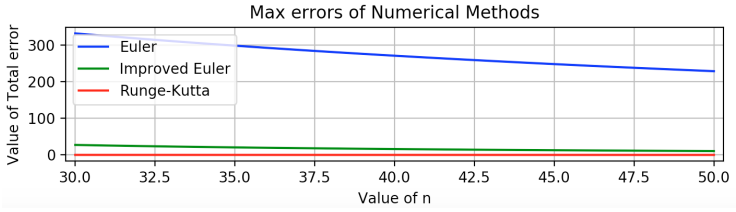
Graphs of methods for initial values(x0 = 1, y0 = 1, X = 7, n = 40):



Local Errors for initial values(x0 = 1, y0 = 1, X = 7, n = 40):

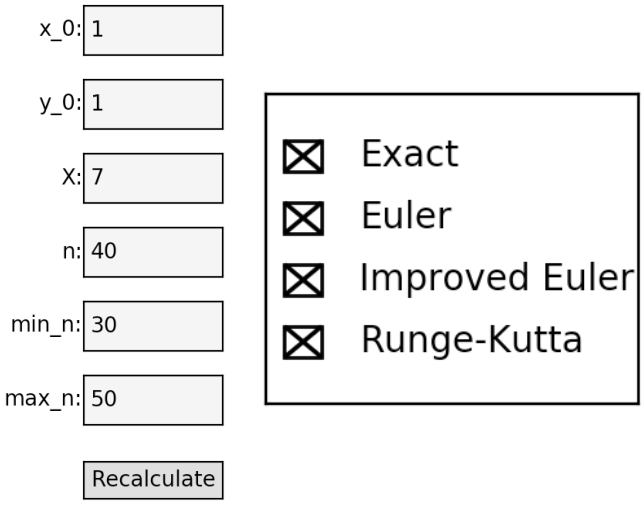


Total Errors for initial values(x0 = 1, y0 = 1, X = 7, nmin = 30, nmax = 50):



Functionality

User can change initial values and by pressing button *Recalculate* get new graphs for updated values. Also user can choose for which methods he/she wants to see graphs in *checkbox*:



4 Analyses

Analyses of difference of approximation in methods
As we can see both from the graphs of solutions and from the errors graphs, for the given differential equation Runge-Kutta Method gives the closest approximation. This conclusion proves our theoretical computations of the approximated local error($O(h^2)$ - Euler's method, $O(h^3)$ - Improved Euler's Method and $O(h^5)$ - Runge-Kutta Method).

Analyses of difference of total errors in methods
As we can guess, with less n the approximation is worse => the total error of methods is increasing, with more n the approximation is better => the total error of methods is decreasing.

5 Source code

You can find the implementation of the project by following the next link
`https://github.com/rufusnufus/DE`