Report

Alina Bogdanova BS18-04

November 2019

1 Exact solution

Problem

IVP №4: $2x^3 + 2\frac{y}{x} = y'$, $x_0 = 1, y_0 = 2$.

Solution

$2x^3 + 2\frac{y}{x} = y'(1)$
$y' - 2\frac{y}{x} = 2x^3(2)$
Consider the complementary equation $y' - 2\frac{y}{x} = 0$
$\frac{\delta y}{\delta x} = 2\frac{y}{x}$
$ln|y| = 2ln|x| + c_2, y \neq 0$
$y = c_1x^2(3)$
Let's return to equation(2):
$c_1'x^2 + 2c_1x - 2c_1x = 2x^3$
As we know$x \neq 0$:
$c_1' = 2x$
$c_1 = x^2 + c(4)$
Let's combine (3) and (4):
$y = x^4 + cx^2$
To solve IVP substitute x_0 and y_0:
$y_0 = x_0^4 + cx_0^2, c = \frac{y_0}{x_0^2} - x_0^2$
$2 = 1 + c \Rightarrow c = 1, y = x^4 + x^2$

Points of discontinuity

The first such point in my IVP is $x = 0$, the second one is $y(x) = 0$ it is reached when $x = 0$, (impossible, case 1) or $(c + x^2) \leq 0$ thus if $c < 0$ roots exist. As we know, $c = \frac{y_0}{x_0^2} - x_0^2$, so y_0 should be greater then x_0^4, otherwise $x = \pm\sqrt{-c}$ is also a point of discontinuity.

2 Implementation details

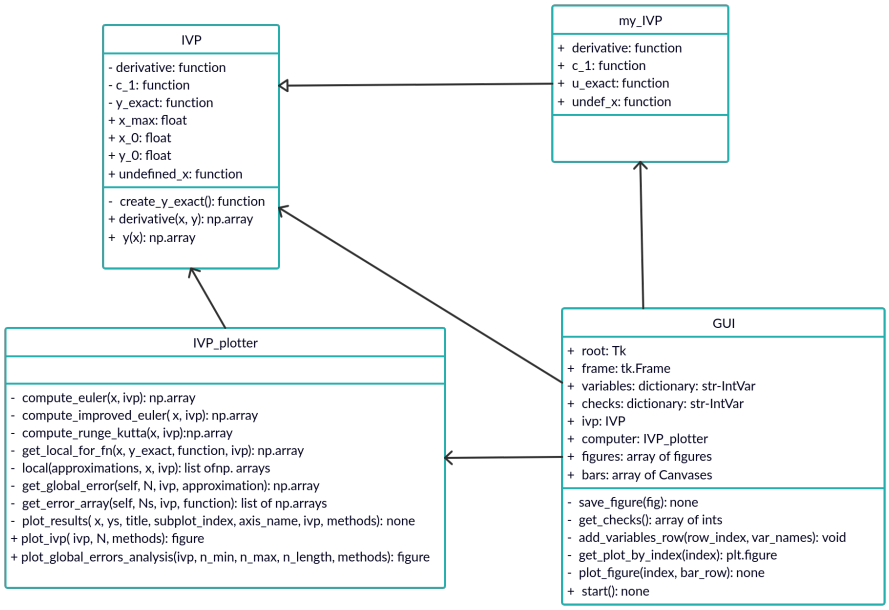

Figure 1: UML Class diagram of the project

Classes

The project has only 4 classes: `IVP`, `my_IVP`, `IVP_plotter` and `GUI`. The first one is an interface from which `my_IVP` is inherited. This way some other problems could be integrated to the system. Such classes as `IVP_plotter` and `GUI` are using only the methods of interface (it's implementation has only initialization). At the figure 1 UML class diagram of the program is shown.

SOLID principles correspondence

Single Responsibility Principle

Each class have it's own responsibility or reason to be changed:

- `IVP` - interface for all tasks
- `my_IVP` - my task
- `IVP_plotter` - plotting
- `GUI` - graphics for desktop app

Open-Closed Principle

This principal means, that classes are open for extension and modification. Is it true in my case? I have an abstraction `IVP` class - interface for any initial value problem. Thus it is possible to extend the work by addition of other IVP's. However, there are also some drawbacks according to this principle in my code: if some features should be added the `GUI` should also be updated. This class is not as flexible, as desired due to the importance of reading the values of many inputs.

Liskov Substitution Principle

This principle says "Derived classes must be substitutable for their base classes"[1]. It meas that everything, that works with the child class should be able to work with ancestor. For my class `IVP_plotter` this property works as for `GUI`. Also error-handling - predecessor of contracts is used in the project (e.g. assertions).

Interface Segregation Principle

Implementation of this principle is not represented in the project due to its small size and tools used.

Dependency Inversion Principle

This principle is similar to the Liskov principle. It requires Interface usage, what is implemented in project by inheritance of `my_IVP` from `IVP` and dependence of all other classes from interface.

The most interesting implementation details

The implementation of the project was divided on two parts: computations and GUI.
Computational part was only about formulas and graphs plotting with the use of such libraries as `numpy` (computations) and `matplotlib`(visualization). The most interesting ideas behind it were:
1. Interface usage
2. Auto-calculating of the constant for IVP
The first one was already discussed, while the second one was reached by usage of a private function `__create_y_exact()`, which returns the function `y(x)` used as exact solution during computations. The code of the method is represented below:

```
def __create_y_exact(self):
        '''
        returns the exact solution of IVP
        with constant substituted
        '''
        c_1 = self.__c_1(self.x_0, self.y_0)
```

```
        return lambda x: self.__y_ex(x, c_1)
}
```

Where `self.__c_1()` is a function, from the input to the `IVP` constructor (it returns the constant of the exact solution), `(x_0, y_0)` - the dot, in which the IVP is defined.
The **GUI part** has much more interesting implementational features.
For visualization of the interface I have used `tkinter` library. In templates it requires a lot of duplicated code to generate the list of the same elements (e.g the list of input labels for such values as x_0, y_0, x_max etc). To except such duplication the method `__add_variables_row()` was used. Its code is represented below:

```
    def __add_variables_row(self, row_index, var_names):
        '''
            row_index - integer
            var_names - list of variables names to be used in Entry
        '''
        col_index = 0
        for var_name in var_names:
            Label(self.frame, text=var_name+':').grid(  row=row_index,
                                                        column=col_index)
            Entry(  self.frame,
                    text=self.variables[var_name]).grid(row=row_index,
                                                        column=col_index+1)
            col_index += 2
```

Where `row_index, self.frame` and `col_index` are the arguments of the location on the GUI window, `self.variables` is a dictionary, where for each variable name the variable of `VarInt` type (special `tkinter` type of variable to control the user's input) is given.
The next important feature of the project - usage of the checkboxes for each type of approximation. This way user can control which graphs should be plotted or and which not.
This feature was also implemented using generation of the similar code using cycles, the variables are also stored as an attributes inside dictionary.
Also in that implementation of the task there is and option for saving of the figure (some buttons are added).
With the help of code optimization the resulted code became quite small: GUI part takes only 87 lines of code, while the computational part requires 150.

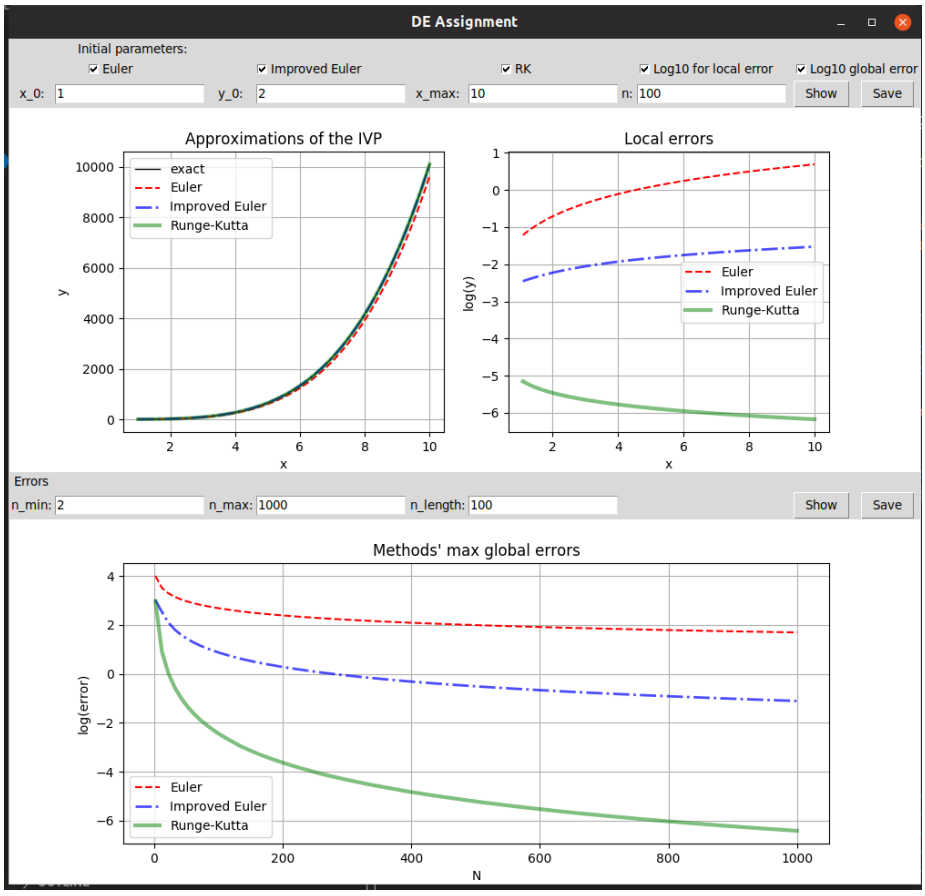

Figure 2: GUI

The resulted GUI is shown on the figure 2

3 Analysis

3.1 Log application

In this part of the report I would like to mention one more implementation detail behind the program: possibility of axis changing.

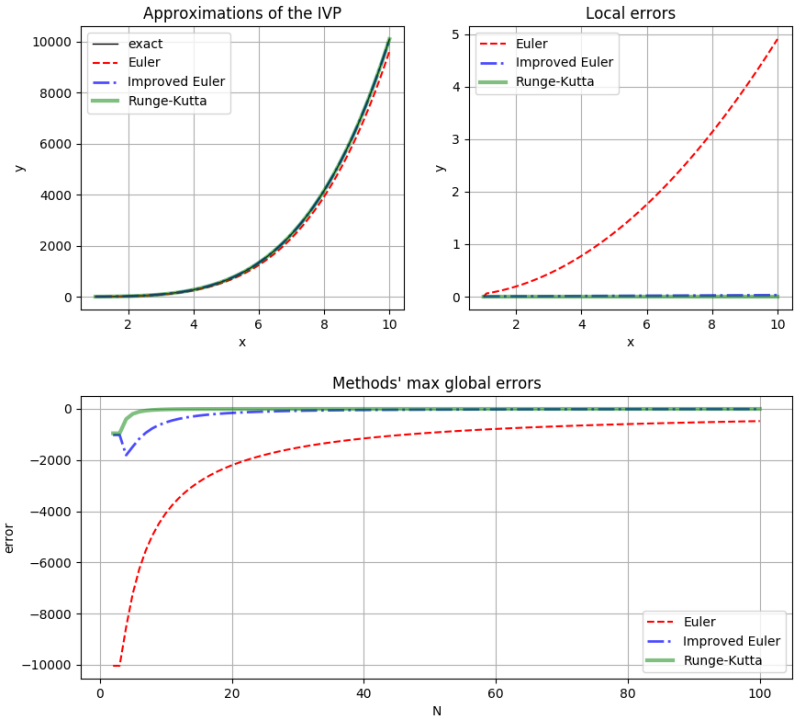

Figure 3: Approximation results, local and global errors for the given IVP without log applied to axis

As it can be seen on the figure 3, if we will plot the errors of the given methods in one plot, the results for Improved Euler and Runge-Kutta methods are not distinguishable. Due to this fact the option of log_{10} function application to the y axis of the error graphs was added.

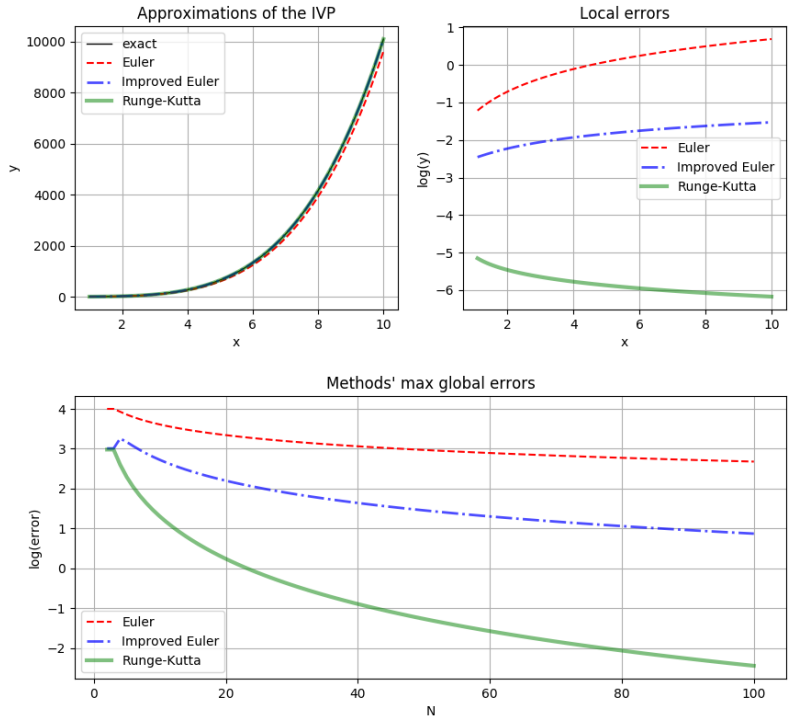

Figure 4: Application of log_{10} to local and global errors for the given IVP

The results of this function application can be seen in the figure 4.

3.2 Errors of the methods

As it can be seen from the figure 4, the local error for Euler method is approximately 100 times greater then Improved Euler method and 10^6 times greater then for Runge-Kutta method.
As for global error, Euler is around 10 times worse then Improved one, and from 10 to 10000 worse then Runge-Kuttas.

4 Result

To sum up, the Runge-Kutta method is the most precise method from Runge-Kutta, Euler and Improved Euler method. The worst method from the given is simple Euler method.
The code is available by the link.

References

[1] SOLID principle explanation

[2] Elementary Differential Equations by William F. Trench. Brooks/Cole Thomson Learning, 2001.