innopolis university

Differential Equations

COMPUTATIONAL PRACTICUM

Student name:
Marko Pezer

Student group:
B18-03

November 2019

CONTENT

I PROBLEM STATEMENT AND THE EXACT SOLUTION OF THE IVP

Problem statement

Find the exact solution of given initial value problem and analyze points of discontinuity, if exist.

$y' = e^y - 2/x$

$y(1) = -2$

$x \in (1, 7)$

Exact solution of the initial value problem

Given equation
$$\frac{dy(x)}{dx} = e^{y(x)} - \frac{2}{x}$$

$$\frac{dy(x)}{dx} = e^{y(x)} - \frac{2}{x}$$

$$e^{y(x)}x - x\frac{dy(x)}{dx} - 2 = 0$$

Let $y(x) = \log(\frac{\mathrm{v(x)}}{\mathrm{x}})$
$$\frac{dy(x)}{dx} = \frac{x\left(\frac{\frac{dv(x)}{dx}}{x} - \frac{v(x)}{x^2}\right)}{v(x)}$$

$$v(x) - \frac{x\frac{dv(x)}{dx}}{v(x)} - 1 = 0$$

Solve for $\frac{dv(x)}{dx}$
$$\frac{dv(x)}{dx} = \frac{(v(x) - 1)v(x)}{x}$$

Divide both sides by $(v(x) - 1)v(x)$
$$\frac{\frac{dv(x)}{dx}}{(v(x) - 1)v(x)} = \frac{1}{x}$$

Integrate both sides with respect to x
$$\int \frac{\frac{dv(x)}{dx}}{(v(x) - 1)v(x)}\, dx = \int \frac{1}{x}\, dx$$

Evaluate the integrals	$\log(-v(x)+1)-\log\big(v(x)\big)=\log(x)+c_1$
Solve for $v(x)$	$v(x)=\dfrac{1}{e^{c_1}x+1}=\dfrac{1}{c_1x+1}$
Substitute back for $y(x)=\log\left(\frac{v(x)}{x}\right)$	$v(x)=xe^{y(x)}$
	$e^{y(x)}x=\dfrac{1}{c_1x+1}$
Solution	$\boldsymbol{y(x)=-\log(c_1x^2+x)}$
By applying initial values, we get	$-2=-\log(c_1+1)$
	$\log(c_1+1)=2$
	$c_1+1=e^2$
	$c_1=e^2-1$
Exact solution of IVP	$\boxed{\boldsymbol{y(x)=-\log\left((e^2-1)\,x^2+x\right)}}$

There are no points of discontinuity in given range.

II IMPLEMENTATION

OOP-design standards

I have chosen C# programming language for implementation. My code is organized within SOLID principles.

The Single Responsibility Principle (SRP)

One class is responsible for only one task.

- **`Form1.cs`**: Main class.
- **`Euler.cs`**: Implementation of the Euler's method.
- **`ImprovedEuler.cs`**: Implementation of the Improved Euler's method.
- **`RungeKutta.cs`**: Implementation of the Runge-Kutta method.
- **`MyEquation.cs`**: Calculation of the exact solution for given equation.
- **`MaxError.cs`**: Calculation of the maximum error for given numerical method depending on number of grid steps.
- **`ChartOne.cs`**: Drawing chart for exact solution and numerical solutions.
- **`ChartTwo.cs`**: Drawing chart for local errors for given numerical method.
- **`ChartThree.cs`**: Drawing graph for total solution.

The Open Closed Principle (OCP)

We can easily extend a class's behavior, without modifying it. Code of my application doesn't have to be changed every time the requirements change.

The Liskov Substitution Principle (LSP)

If some module of application is using a main class then the reference to that main class can be replaced with a derived class without affecting the functionality of the module. New derived classes are extending the base classes without changing their behavior.

The Interface Segregation Principle (ISP)

My code consists of two interfaces that are client specific.

- **`NumericalMethod.cs`**: This interface represents generalization of all numerical methods classes. Classes **`Euler.cs`**, **`ImprovedEuler.cs`**, and **`RungeKutta.cs`** inherit from this interface.

- **`Plotting.cs`**: This interface represents generalization of all classes for plotting graphs. Classes **`ChartOne.cs`**, **`ChartTwo.cs`**, and **`ChartThree.cs`** inherit from this interface.

Dependency Inversion Principle

In my code, entities depend on abstractions, not on concretions. As well, abstractions do not depend on details.

UML Class Diagram

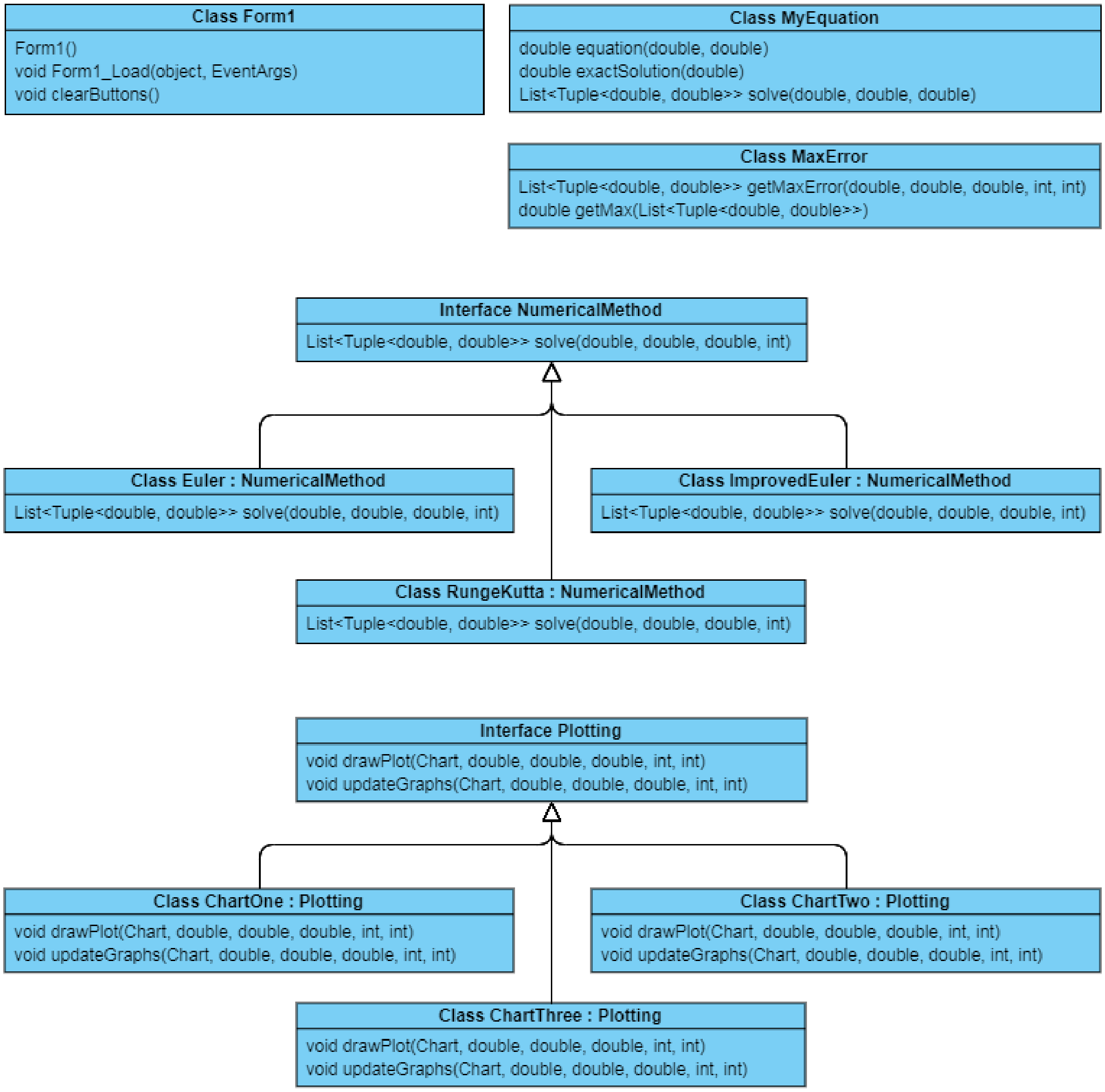

Image 1: *UML diagram of classes*

Implementation of numerical methods

Each method gets same values for x and produces approximations on these points. The precision of a particular method depends on number of grid steps.

Euler's method implementation

```
public List<Tuple<double, double>> solve(double X0, double Y0, double UPPER_BOUND, int
num_segments)
        {
            List<Tuple<double, double>> Points = new List<Tuple<double, double>>();

            double step = (UPPER_BOUND - X0) / num_segments;
            double x_curr = X0;
            double y_curr = Y0;

            Points.Add(Tuple.Create(X0, Y0));

            for (int i = 0; i < num_segments; i++)
            {
                y_curr = y_curr + step * myEq.equation(x_curr, y_curr);
                x_curr = x_curr + step;
                Points.Add(Tuple.Create(X0 + (i + 1) * step, y_curr));
            }

            return Points;
        }
```

Improved Euler's method implementation

```
public List<Tuple<double, double>> solve(double X0, double Y0, double UPPER_BOUND, int
num_segments)
        {
            List<Tuple<double, double>> Points = new List<Tuple<double, double>>();

            double step = (UPPER_BOUND - X0) / num_segments;
            double x_curr = X0;
            double y_curr = Y0;

            Points.Add(Tuple.Create(X0, Y0));

            for (int i = 0; i < num_segments; i++)
            {
                double k1 = myEq.equation(x_curr, y_curr);
                double k2 = myEq.equation(x_curr + step, y_curr + step * k1);
                y_curr = y_curr + step * (k1 + k2) / 2;
                x_curr = x_curr + step;
                Points.Add(Tuple.Create(X0 + (i + 1) * step, y_curr));
            }

            return Points;
        }
```

Runge-Kutta method implementation

```
public List<Tuple<double, double>> solve(double X0, double Y0, double UPPER_BOUND, int
num_segments)
        {
            List<Tuple<double, double>> Points = new List<Tuple<double, double>>();

            double step = (UPPER_BOUND - X0) / num_segments;
            double x_curr = X0;
            double y_curr = Y0;

            Points.Add(Tuple.Create(X0, Y0));

            for (int i = 0; i < num_segments; i++)
            {
                double k1 = myEq.equation(x_curr, y_curr);
                double k2 = myEq.equation(x_curr + step / 2, y_curr + step / 2 * k1);
                double k3 = myEq.equation(x_curr + step / 2, y_curr + step / 2 * k2);
                double k4 = myEq.equation(x_curr + step, y_curr + step * k3);
                y_curr = y_curr + step * (k1 + 2 * k2 + 2 * k3 + k4) / 6;
                x_curr = x_curr + step;
                Points.Add(Tuple.Create(X0 + (i + 1) * step, y_curr));
            }

            return Points;

        }
```

III PLOTTING ANALYSIS

Exact solution and numerical solution plot

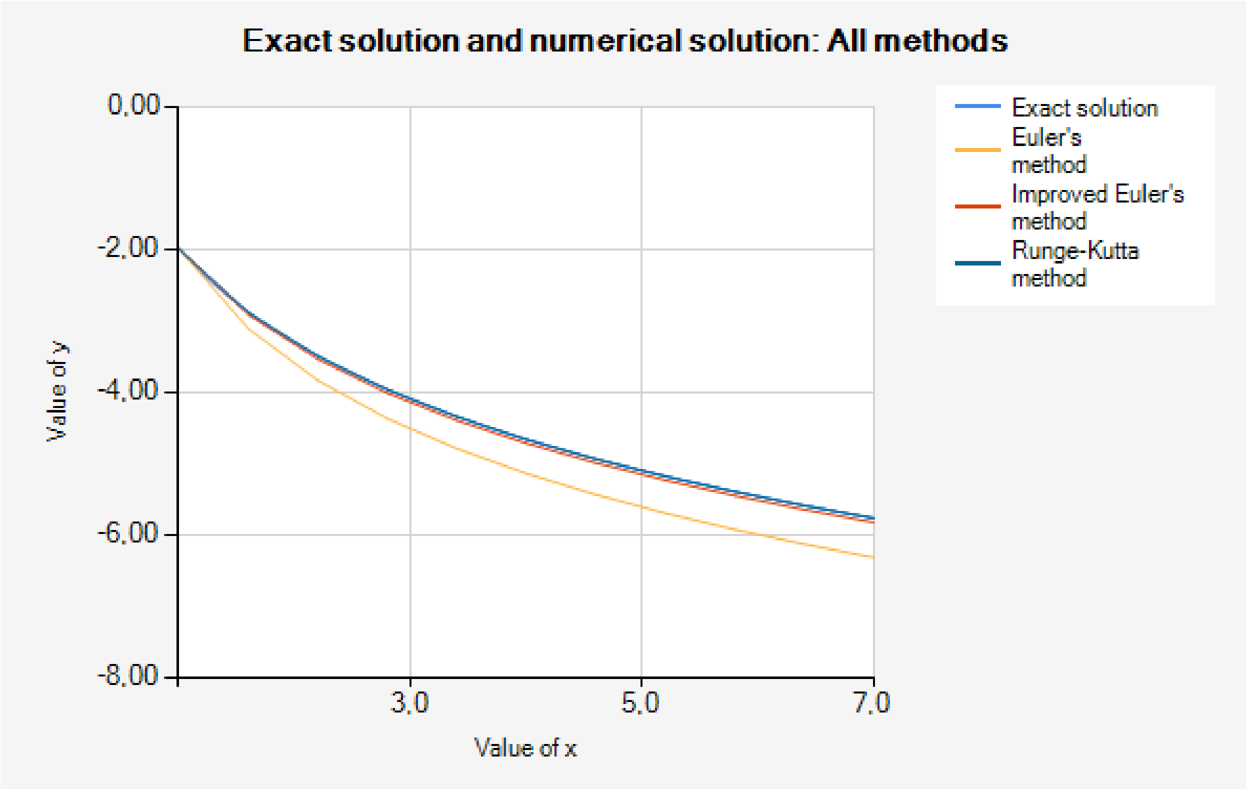

Image 2: *Exact solution and numerical solutions* $(x_0 = 1, y_0 = -2, X = 7,\ N = 10)$

On Image 2 we can see the curve of the exact solution and the curves of numerical solutions for all three methods. From this plot, we conclude that Runge-Kutta method is most accurate (relative error is $< 0.04\%$ over the interval), while typical Euler method gives the least precise results (relative error is $< 9\%$ over the interval).

Local errors for numerical methods plot

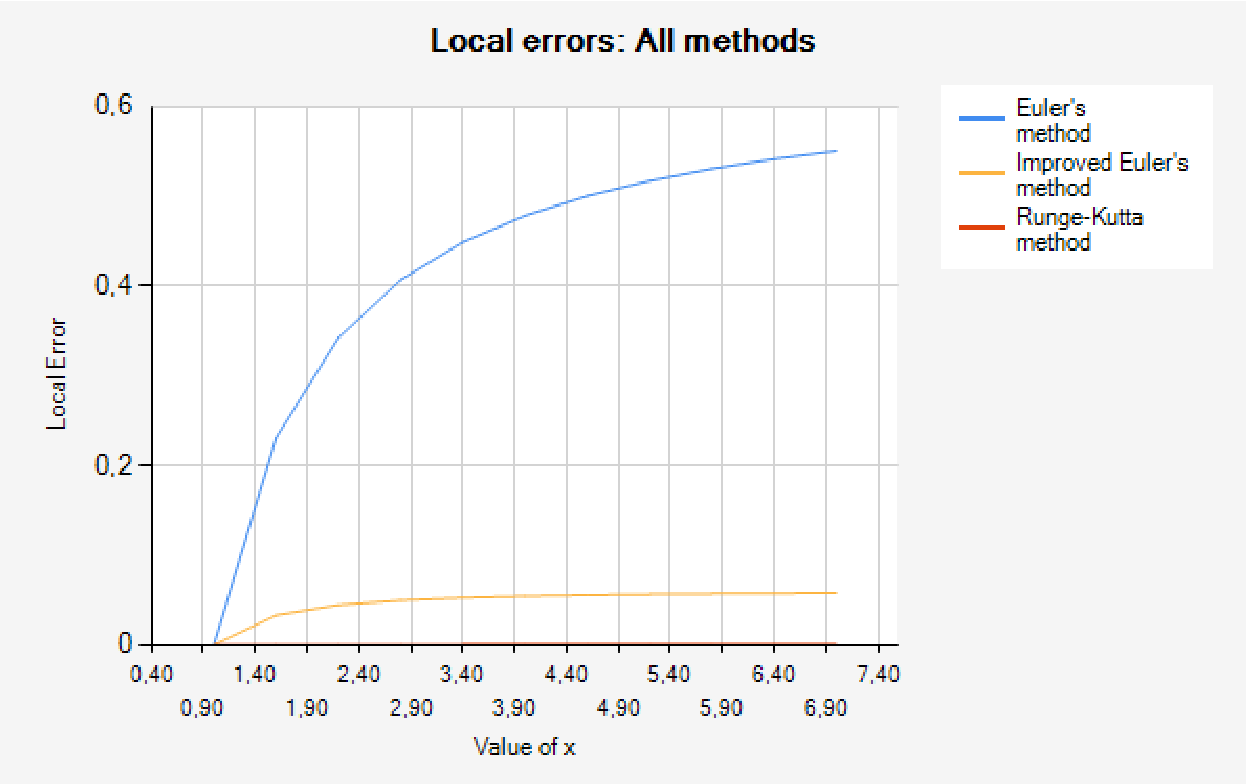

Image 3: *Local errors for numerical methods* $(x_0 = 1, y_0 = -2, X = 7,\ N = 10)$

On Image 3 we have plot of local errors for numerical methods. We can easily compare methods using this plot. The curve of Runge-Kutta method lies on x-axis because error of this method is almost 0.

For example, we can see that for $x = 7$ error of Euler's method is approximately 0.55, error of Improved Euler's method is approximately 0.06, while error of Runge-Kutta method is almost 0.

Total errors for numerical methods plot

On images 4, 5, and 6, we have respectively total errors plots for Euler's method, Improved Euler's method and Runge-Kutta method depending on number of grid steps (from 1 to 50). Application allows us to easily change boundaries of grid steps numbers.

As we can see from plots, for small number of steps (1-5) all methods have very big errors:

- Euler: > 7,
- Improved Euler: > 2.5,
- Runge-Kutta: > 0.3.

On the other hand, for big number of steps (greater than 40), we have really small errors:

- Euler: < 0.1,
- Improved Euler: < 0.01,
- Runge-Kutta: < 0.0001.

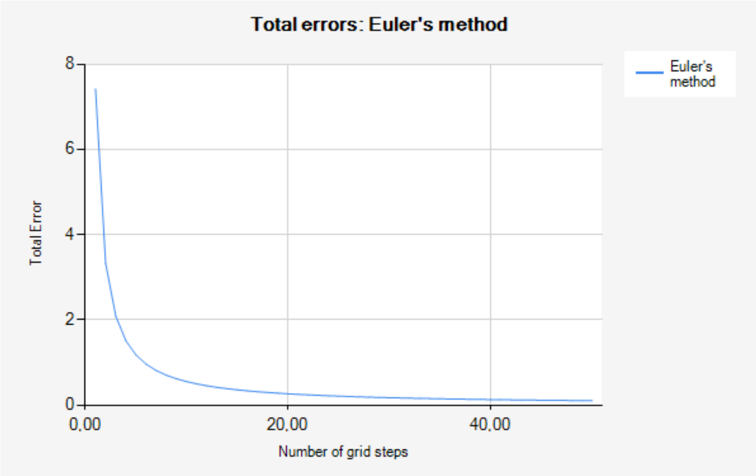

Image 4: *Total error plot for Euler's method*

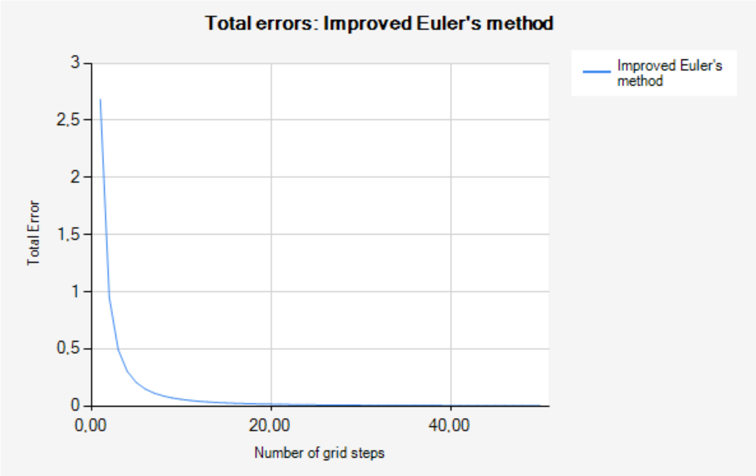

Image 5: *Total error plot for Improved Euler's method*

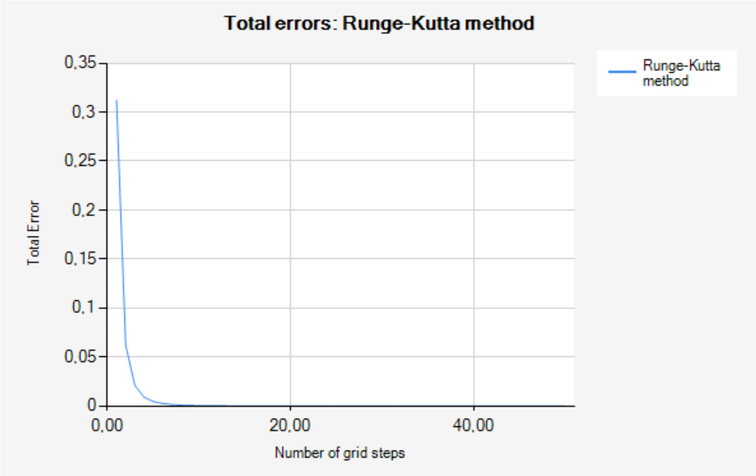

Image 6: *Total error plot for Runge-Kutta method*