Charikova Mariia
BS2-7 group

Differential Equations
Computational Practicum Assignment

Variant 13

Given equation: $y' = sin^2(x) + y \cdot cot(x)$

Solution

First order linear Ordinary Differential Equation.

$$y' - \cot(x)y = \sin^2(x)$$

Find the integrating factor: $\mu(x) = \frac{1}{\sin(x)}$

$$\frac{1}{\sin(x)}y' - \frac{1}{\sin(x)}\cot(x)y = \frac{1}{\sin(x)}\sin^2(x)$$

$$\frac{1}{\sin(x)}y' - \frac{1}{\sin(x)}\cot(x)y = \sin(x)$$

$$\left(\frac{1}{\sin(x)}y\right)' = \sin(x)$$

$$\frac{1}{\sin(x)}y = \int \sin(x)dx$$

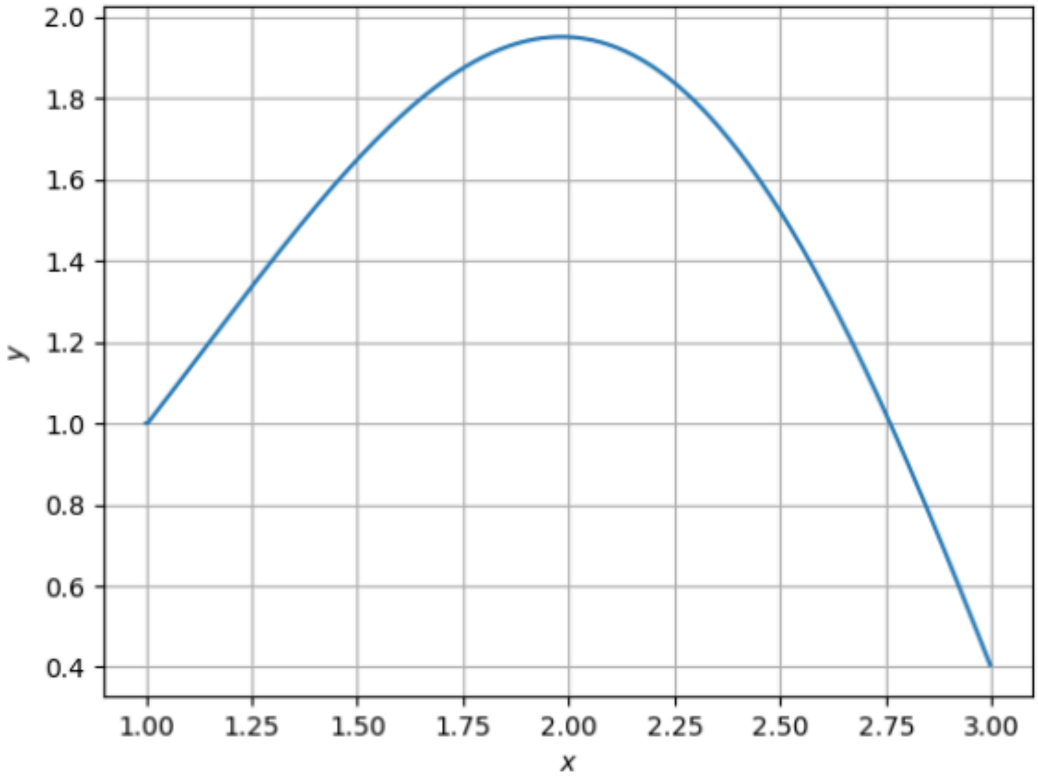
$$\frac{1}{\sin(x)}y = -\cos(x) + c_1$$

$$\boxed{y = -\cos(x)\sin(x) + c_1\sin(x)}$$ - *general solution*

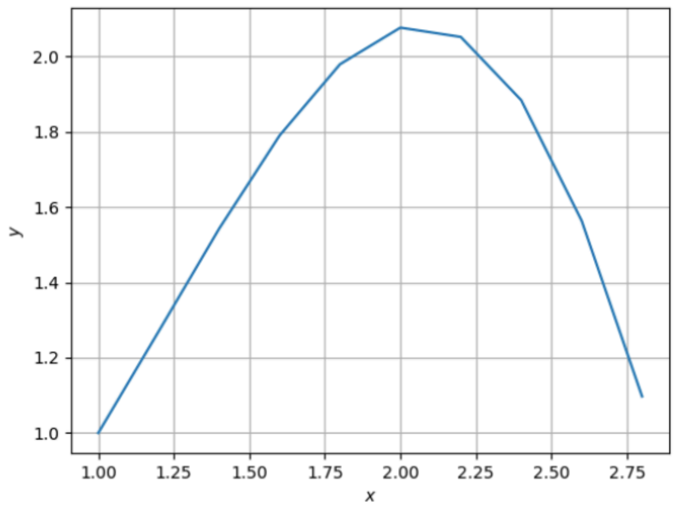
$c = \frac{y}{sin(x)} + cos(x)$ With initial values x0 = 1 and y0 = 1, c = 1.73

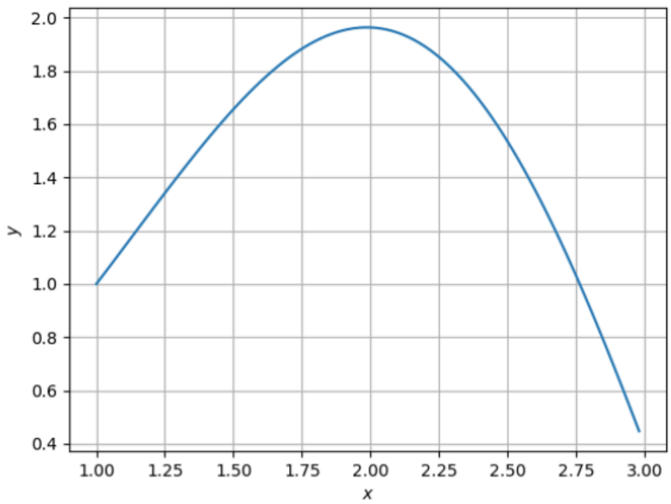
Total approximation analysis

1. Exact Solution

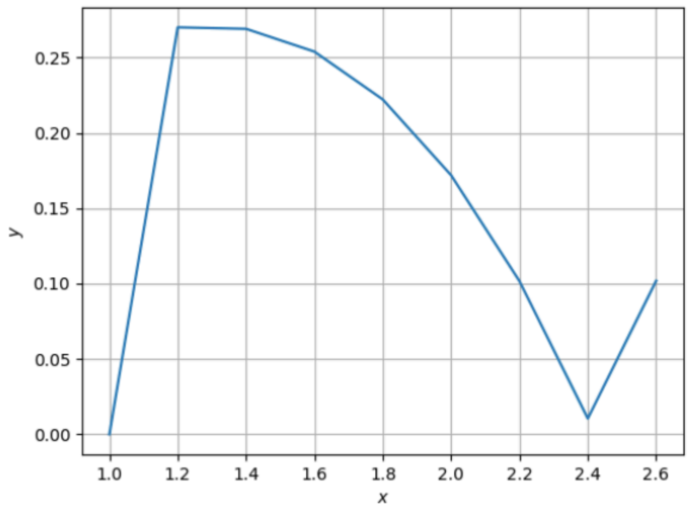


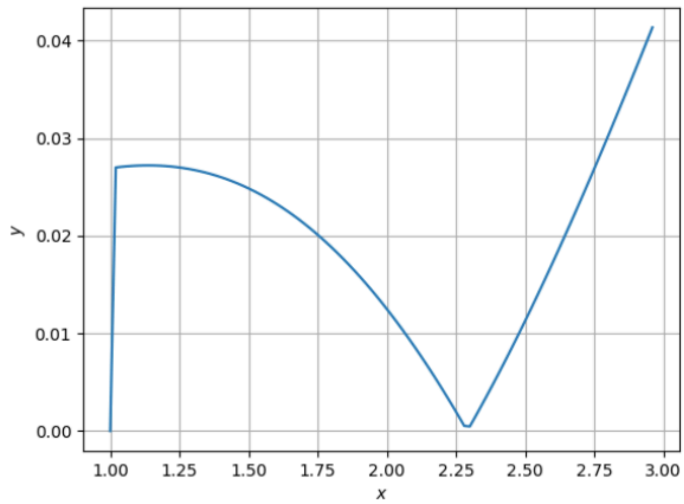
2.Euler’s Method approximation (10/100 steps)



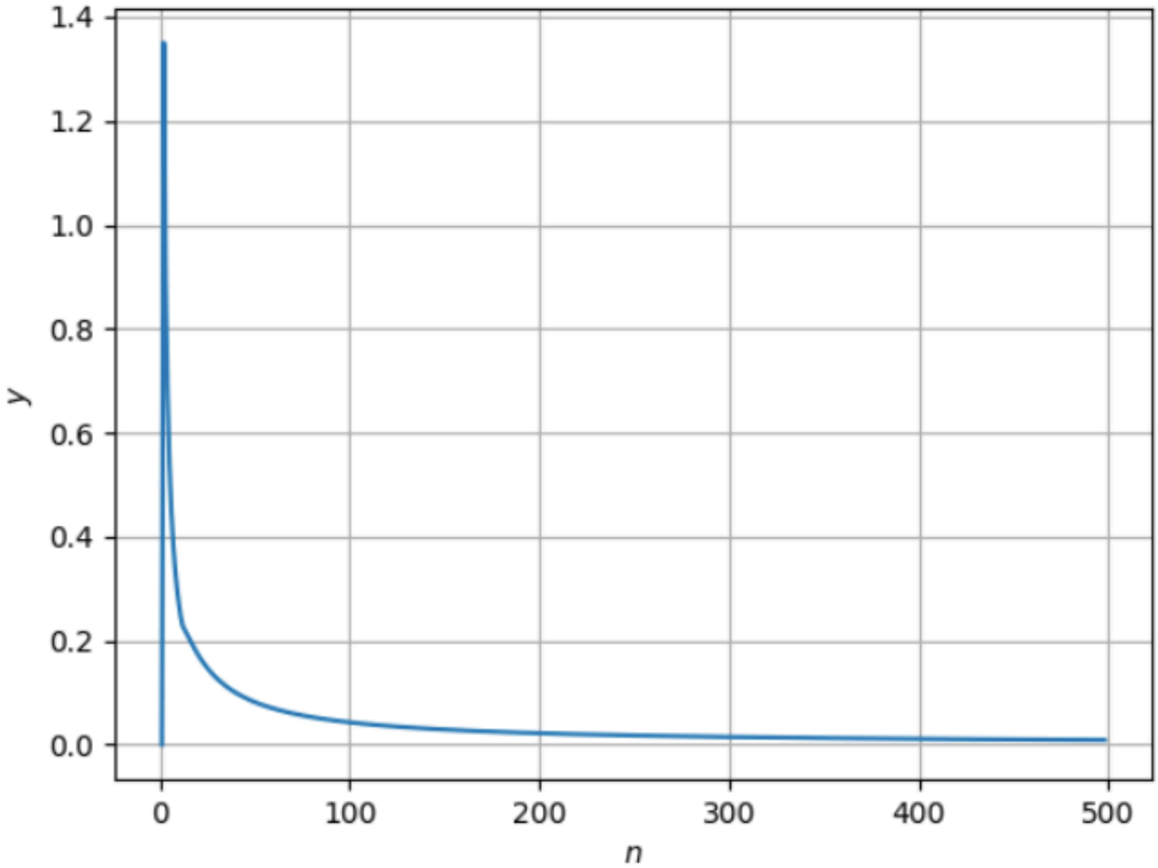


3. Euler's Method local errors (10/100 steps)

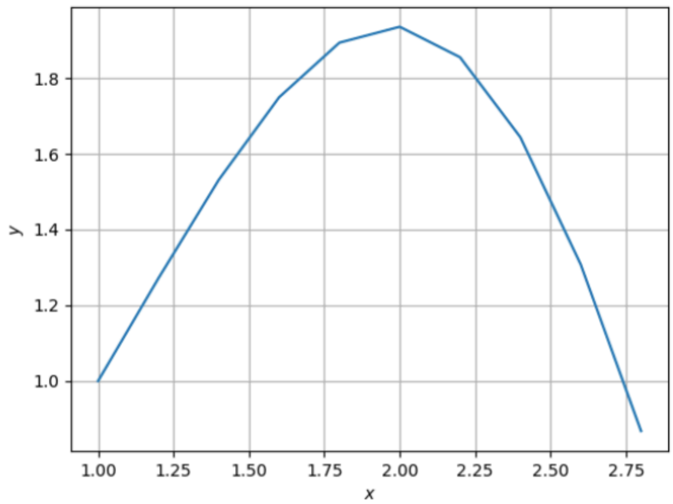


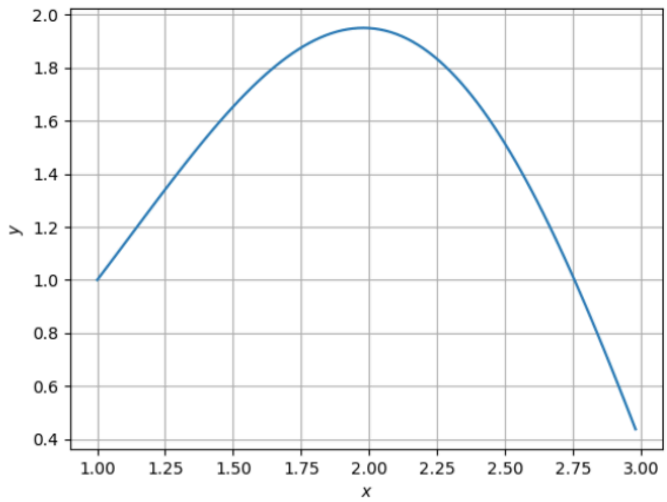


4.Euler's Method global errors (<500 steps)

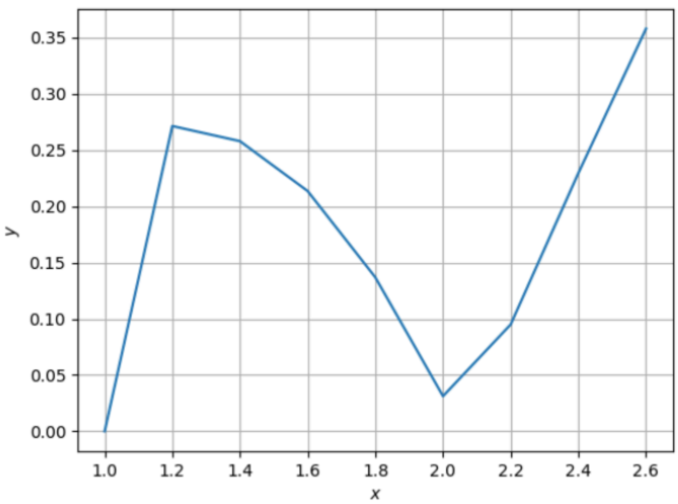


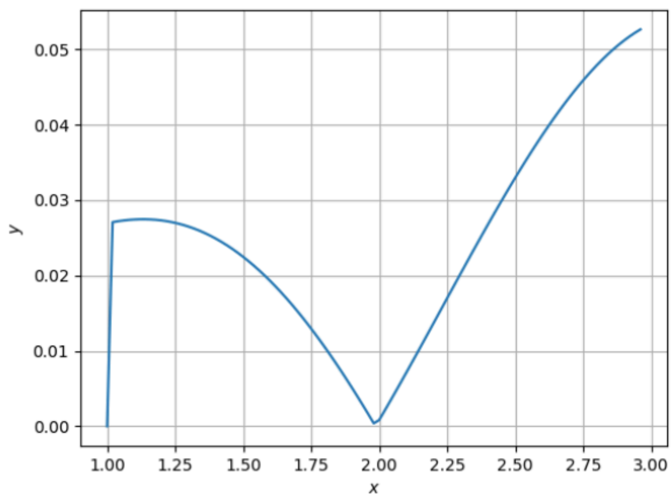
5. Improved Euler's Method approximation (10/100 steps)



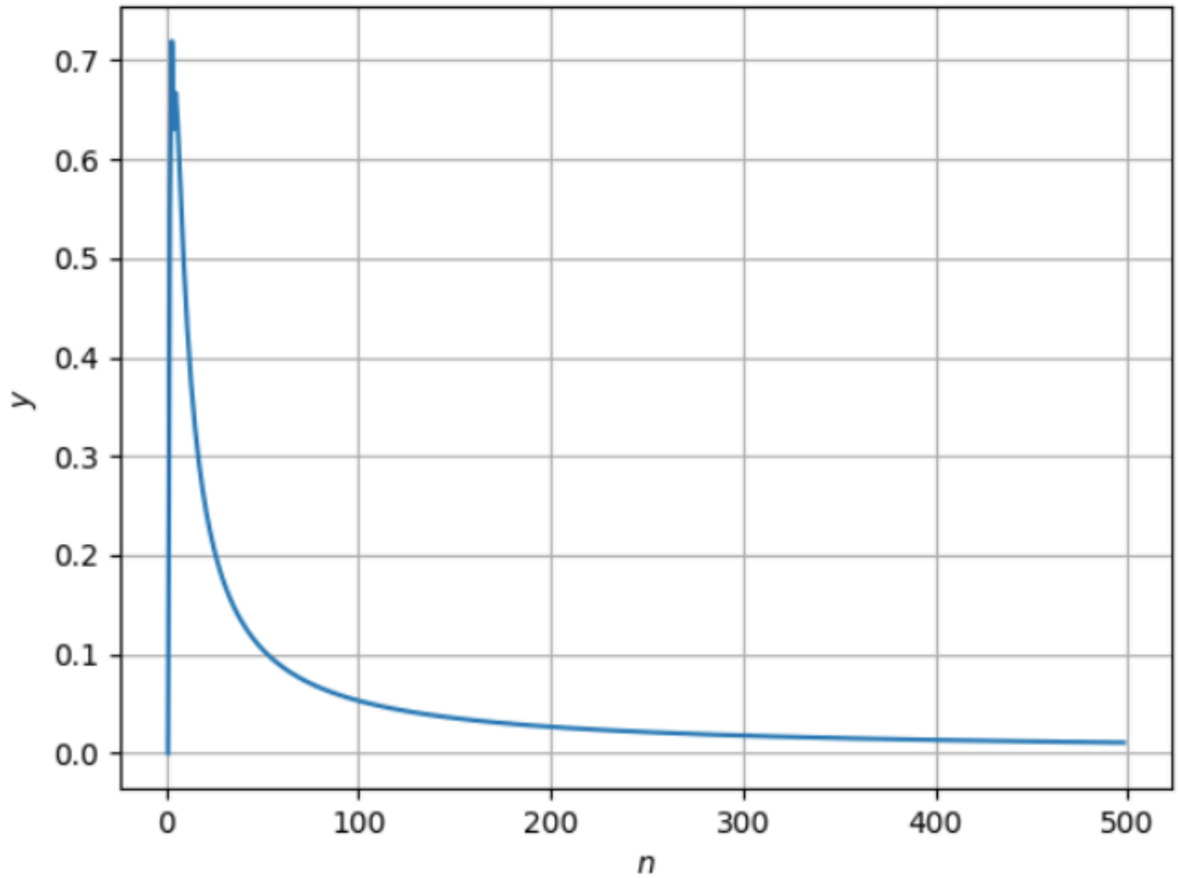


6. Improved Euler's Method local errors (10/100 steps)

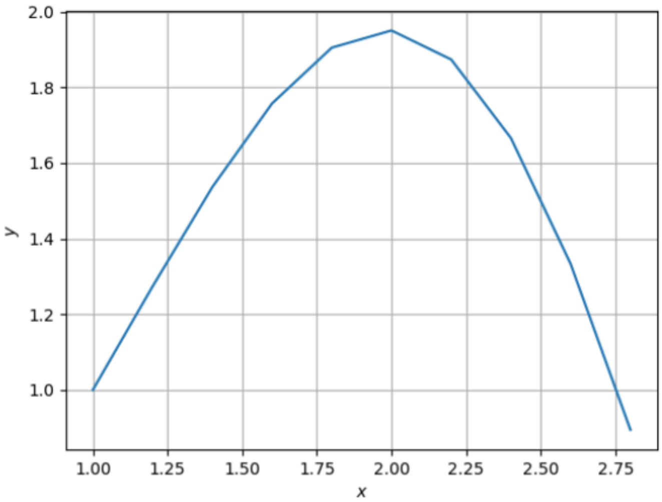


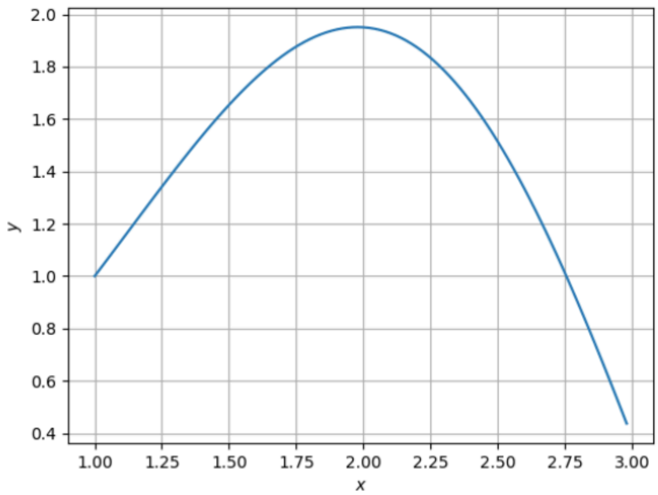


7. Improved Euler's Method global errors (<500 steps)

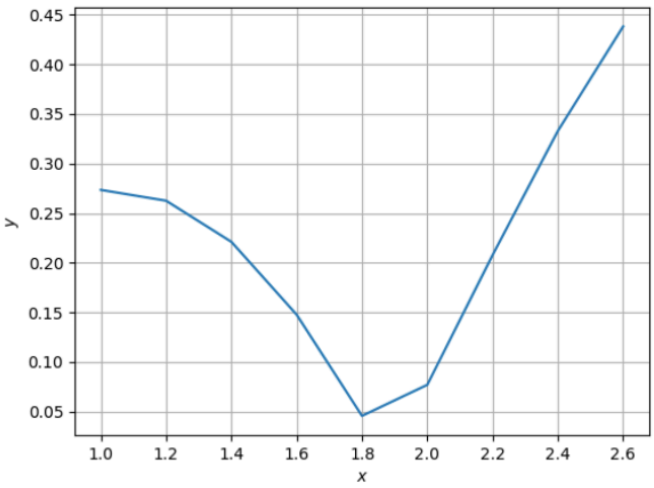


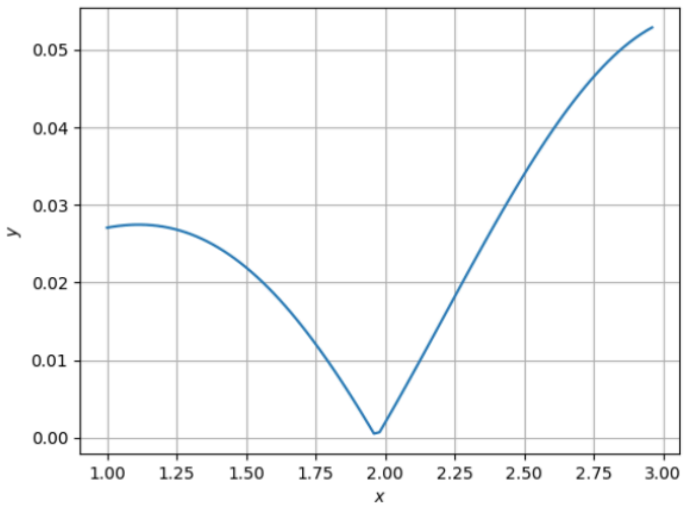
8. Runge-Kutta's Method approximation (10/100 steps)



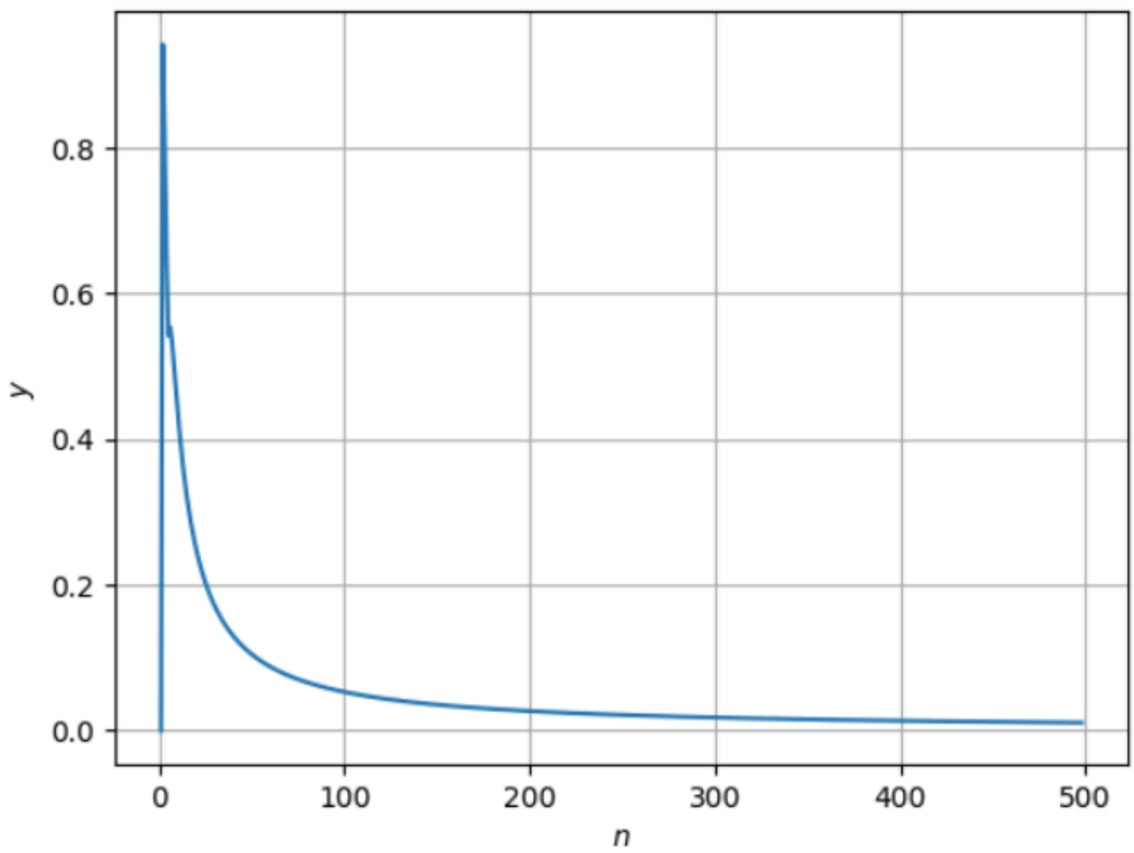


9. Runge-Kutta’s Method local errors (10/100 steps)

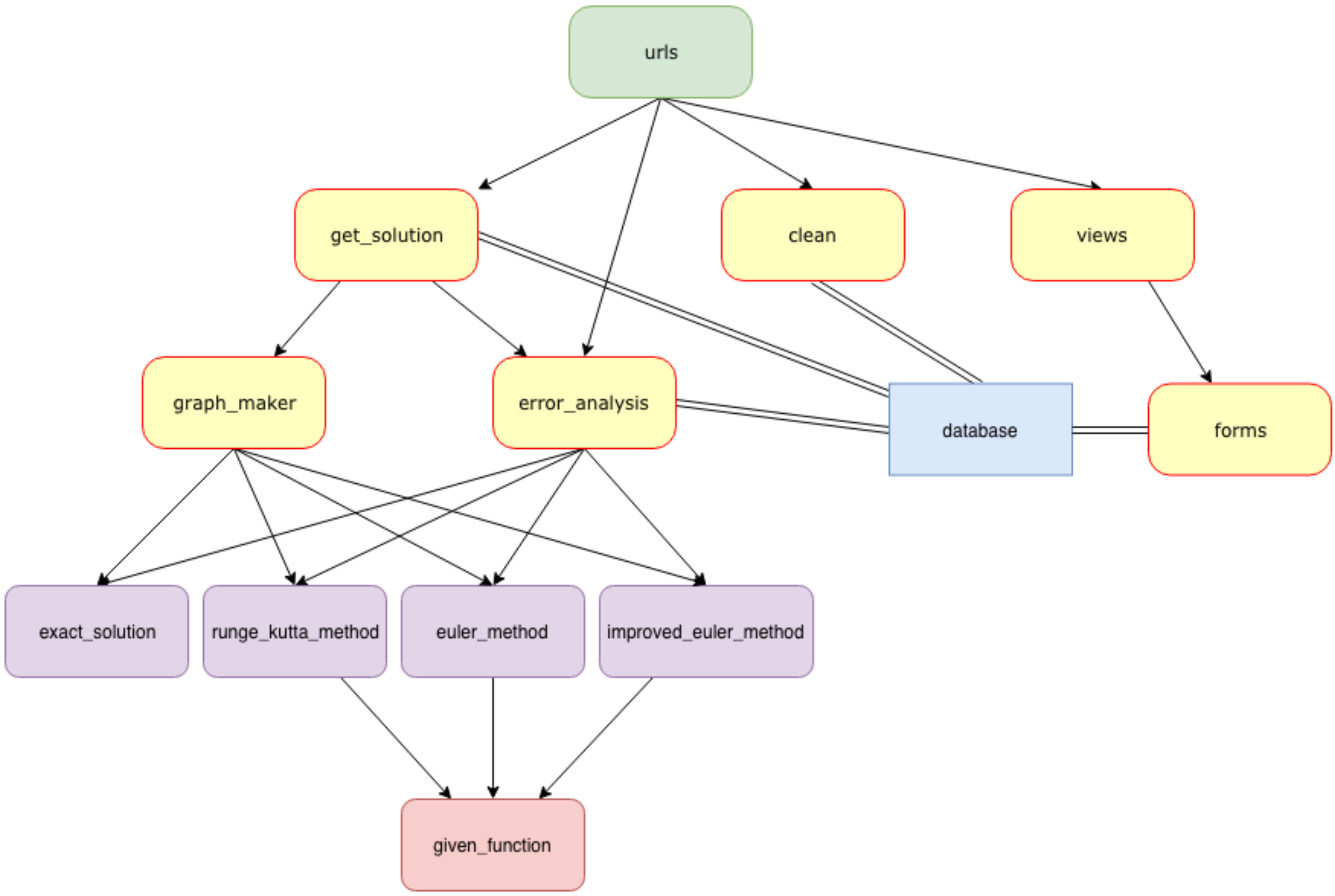




10. Runge-Kutta’s Method global errors (<500 steps)



UML class diagram

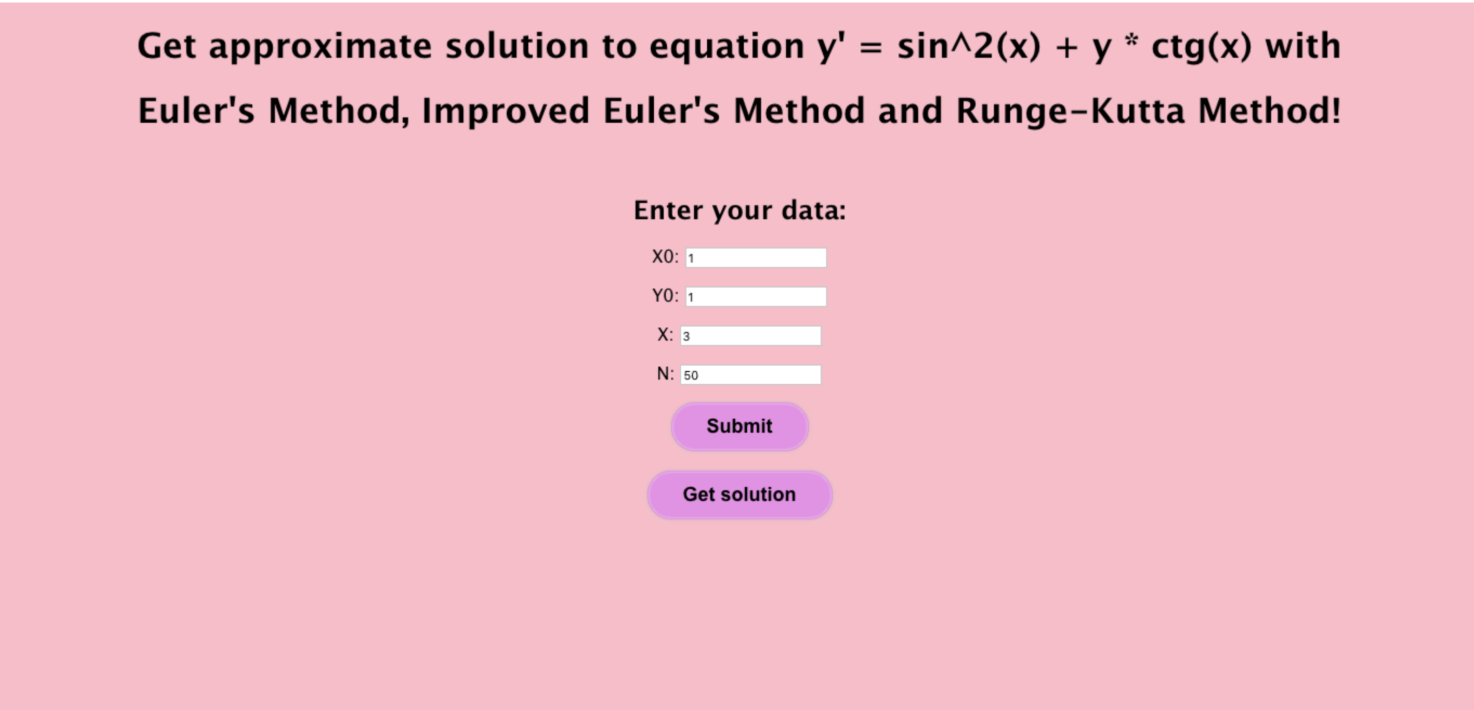


- Web application requires class **urls** for managing all html pages and function which are calling by redirection.
- Starting page contains the form. In the class **views** we save values from the form to the database.
- Class **forms** is a template for form for adding values to the database.
- In class **get_solution** there are parsing of values from the database, constructions for creating web page with approximate solutions using different methods and local errors and calling to the 2 functions to make a graphs and analyse global errors.
- Class **graph_maker** creates graphs of solutions and local errors using matplotlib python library and takes data from classes of different methods.
- In class **exact_solution** we use solution which have been calculated by hands. There are filling the arrays of function values according to the different x-es using exact solution.
- In classes **euler method**, **improved_euler_method** and **runge_kutta_method** we use special formulas (different for each method) and class of given function to fill arrays with y-es according to different x-es, like in the exact solution but not as precisely as in exact solution.

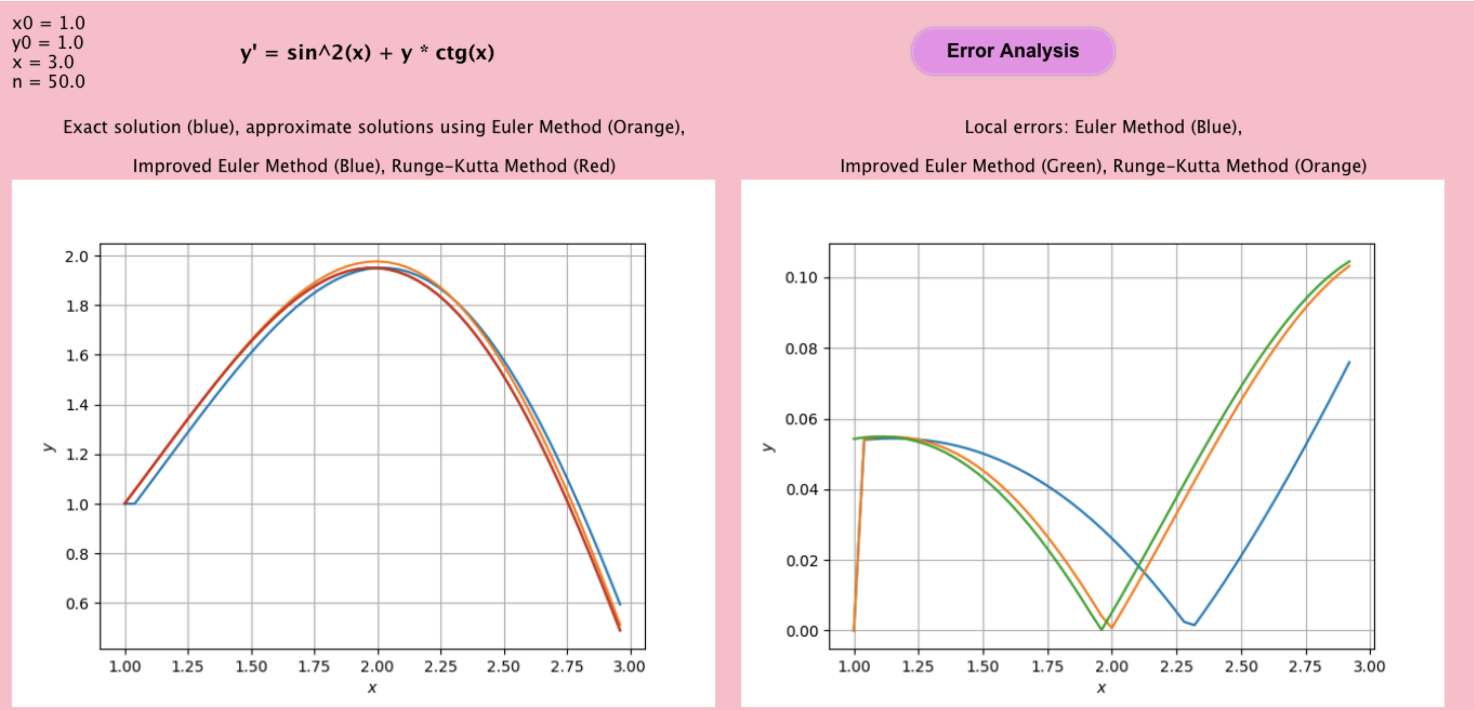
- Class **error_analysis** is for analysing maximum global errors of approximate solutions with different methods according to the different amount of steps.
- In class **clean** we delete all of data in database.

GUI

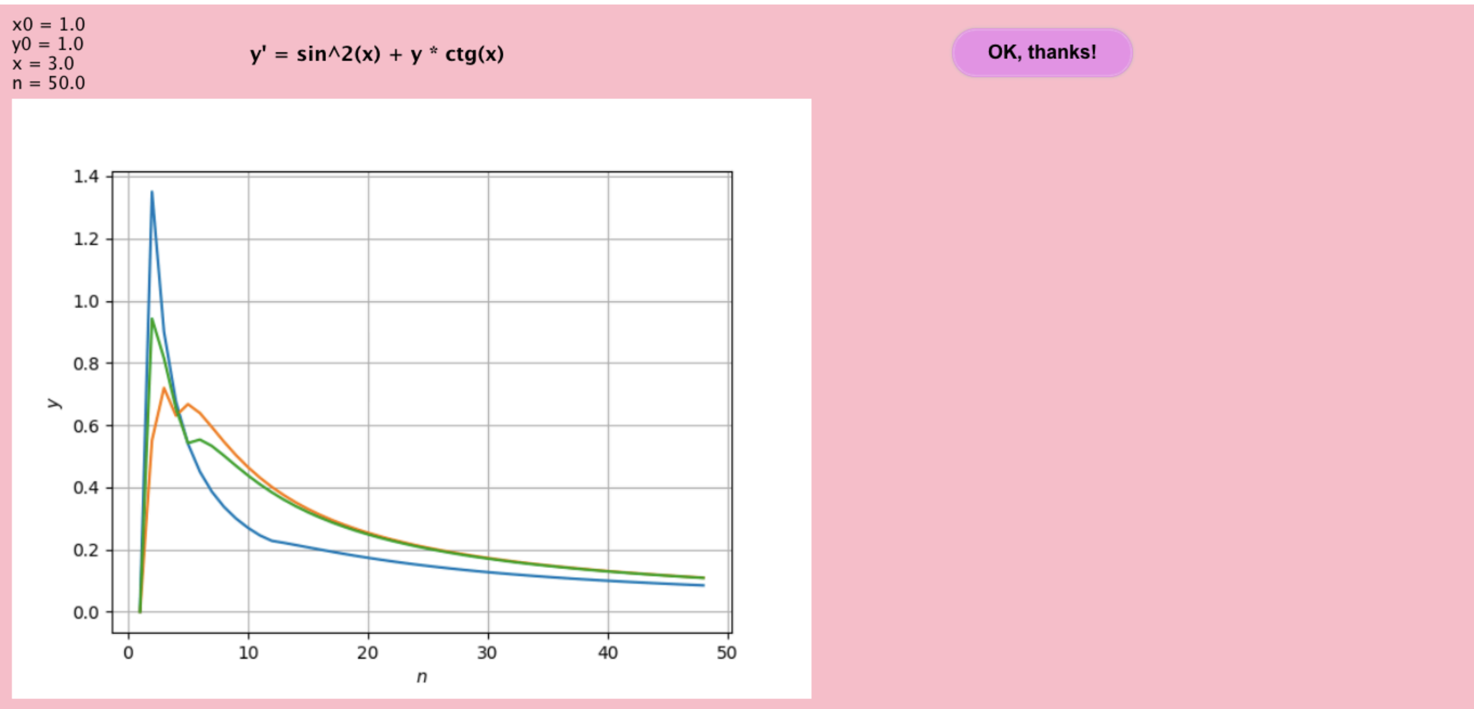
1. **Starting page.**
Form with entering data for approximate solution



2.**Solution page**.
Graph with approximate solutions using Euler's method, Improved Euler's Method, Runge-Kutta Method and Exact solution (on the left). Graph with modulus of local errors of these three methods (on the right).



3.**Error Analysis page.**
Graph with decreasing number of maximum global errors of approximate solutions using Euler's method, Improved Euler's Method, Runge-Kutta Method



Interesting parts of code

Given function

```
def function_for_computation(x, y):
    return sin(x) * sin(x) + (y * (cos(x) / sin(x)))
```

Exact Solution

```
def computations(x0, y0, x, n):
    ys_exact = []
    xs_exact = []
    n = int(n)
    h = (x - x0) / n
    xs_exact.append(x0)
    ys_exact.append(y0)
    c = y0/(sin(x0)) + cos(x0)
    for i in range(n - 1):
        x = xs_exact[-1]
        y = c * sin(x) - (sin(x) * cos(x))
        x += h
        xs_exact.append(x)
        ys_exact.append(y)
    return xs_exact, ys_exact
```

Euler's Method Solution

```
def computations(x0, y0, x, n):
    xs = []
    ys = []
    xs.append(x0)
    ys.append(y0)
    n = int(n)
    h = (x - x0) / n
    for i in range(n - 1):
        x = xs[-1]
        y = ys[-1]
        x_next = x + h
        f = given_function.function_for_computation(x, y)
        y_next = y + h * f
        xs.append(x_next)
        ys.append(y_next)
    return xs, ys
```

Improved Euler's Method Solution

```
def computations(x0, y0, x, n):
    xs = []
    ys = []
    xs.append(x0)
    ys.append(y0)
    n = int(n)
    h = (x - x0) / n
    for i in range(n - 1):
        x = xs[-1]
        y = ys[-1]
        x_next = x + h
        f1 = given_function.function_for_computation(x, y)
        f2 = given_function.function_for_computation(x + h, y + h
* given_function.function_for_computation(x, y))
        y_next = y + h * ((f1 + f2) / 2)
        xs.append(x_next)
        ys.append(y_next)
    return xs, ys
```

Runge-Kutta's Method Solution

```
def computations(x0, y0, x, n):
    xs = []
    ys = []
    xs.append(x0)
    ys.append(y0)
    n = int(n)
    h = (x - x0) / n
    for i in range(n - 1):
        x = xs[-1]
        y = ys[-1]
        k1 = h * given_function.function_for_computation(x, y)
        k2 = h * given_function.function_for_computation(x + h/2,
y + k1/2)
        k3 = h * given_function.function_for_computation(x + h/2,
y + k2/2)
        k4 = h * given_function.function_for_computation(x + h, y
+ k3)
        x_next = x + h
        y_next = y + k1/6 + k2/3 + k3/3 + k4/6
        xs.append(x_next)
        ys.append(y_next)
    return xs, ys
```