Computational problem

By *Timur Mustafin*, BS17-03

Solving original equation:

$y' = y^2e^x + 2y$ - This is a first-order Bernoulli o. d. e.

Transforming into the form $y' + p(x)\,y = q(x)\,y^n$, we get:

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

$z = y^{1-n} = y^{1-2} = y^{-1}$

$-z' - 2z = e^x$

$z' + 2z = -e^x$

Solving complementary equation:

$z_c' + 2z_c = 0$

$-\frac{z_c'}{2z_c} = 1$

$\int -\frac{1}{2y} dz_c = \int 1dx$

$-\frac{1}{2}\ln(z_c) = x$

$z_c = e^{-2x}$ - Solution of the complementary equation

$z = uz_c = ue^{-2x}$

$u'e^{-2x} = -e^x$

$u' = -e^{3x}$

$u = \int -e^{3x}dx$

$u = -\frac{1}{3}e^{3x} + c_1$

$z = (-\frac{1}{3}e^{3x} + c_1)e^{-2x} = \frac{-\frac{1}{3}e^{3x}+c_1}{e^{2x}}$

Substituting back:

$y^{-1} = \frac{-\frac{1}{3}e^{3x}+c_1}{e^{2x}}$

$y = \frac{e^{2x}}{-\frac{1}{3}e^{3x}+c_1}$

$y = \frac{3e^{2x}}{-e^{3x}+c_1}$

Initial value: y(1)=0.5

$\frac{1}{2} = \frac{3e^{2 \cdot 1}}{-e^{3 \cdot 1} + c_1}$

$c_1 = 6e^2 + e^3$

Plugging in c_1:

$y = \frac{3e^{2x}}{-e^{3x} + 6e^2 + e^3}$

Implementation

Math part of implementation

Each method represents as a function with arguments x0, y0 and x(right limit on Ox axis).

Reference solution

```
def reference_solution(self, x0, y0, x):
    # Calculating constant value for given equation
    def const_function(x, y):
        return 3 * math.exp(2 * x) / y + math.exp(3 * x)
    # Original equation
    def my_function(x, constant):
        return (3 * math.exp(2 * x)) / (constant - math.exp(3 * x))

    # prevent calculating constant each time we call my_function()
    const = const_function(x0, y0)

    x = [i for i in np.arange(x0, x + self.DELTA, self.DELTA)]
    y = []

    # calculation
    for i, v in enumerate(x):
        value = my_function(v, const)

        y.insert(i, value)

    return x, y
```

Euler method

According to formula from well-know website (https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%AD%D0%B9%D0%BB%D0%B5%D1%80%D0%B0#%D0%9E%D0%BF%D0%B8%D1%81%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B0):

$$y_i = y_{i-1} + (x_i - x_{i-1})f(x_{i-1}, y_{i-1}), \quad i = 1, 2, 3, \dots, n$$

```
def euler(self, x0, y0, x):
        x = [i for i in np.arange(x0, x + self.DELTA, self.DELTA)]
        y = [y0]

        for i, v in enumerate(x):
            # first one is just an IVP
            if i == 0:
                continue
            value = y[i - 1] + self.DELTA * self.f(x[i - 1], y[i - 1])
            y.insert(i, value)

        # just to make sure that dimensions are the same
        x, y = x[:len(y)], y[:len(x)]

        return x, y
```

Improved Euler method

From the same well-know website
(https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%AD%D0%B9%D0%BB%D0%B5%D1%80%D0%B0#%D0%9C%D0%BE%D0%B4%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D0%B8_%D0%B8_%D0%BE%D0%B1%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D1%8F)

$$y_i = y_{i-1} + (x_i - x_{i-1})\frac{f(x_{i-1},y_{i-1})+f(x_i,y_{i-1}+(x_i-x_{i-1})f(x_{i-1},y_{i-1})._i)}{2}$$

```
def improved_euler(self, x0, y0, x):
    x = [i for i in np.arange(x0, x + self.DELTA, self.DELTA)]
    y = [y0]

    for i, v in enumerate(x):
        # first one is just an IVP
        if i == 0:
            continue

        # according to formula
        value = y[i - 1] + self.DELTA / 2 * (
                self.f(x[i - 1], y[i - 1]) + self.f(x[i], y[i - 1] + self.DELTA * se
        y.insert(i, value)

    # just to make sure that dimensions are the same
    x, y = x[:len(y)], y[:len(x)]

    return x, y
```

Runge-Kutta method

Again, from the favorite students' website
(https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods#The_Runge%E2%80%93Kutta_method)

$$y_{n+1} = y_n + \tfrac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right),$$
$$x_{n+1} = x_n + h$$

$$k_1 = hf(x_n, y_n),$$
$$k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right),$$
$$k_3 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right),$$
$$k_4 = hf\left(x_n + h, y_n + k_3\right).$$

```
def runge_kutta(self, x0, y0, x):
    def calculate(x, y):
        k1 = self.DELTA * self.f(x, y)
        k2 = self.DELTA * self.f(x + self.DELTA / 2, y + k1 / 2)
        k3 = self.DELTA * self.f(x + self.DELTA / 2, y + k2 / 2)
        k4 = self.DELTA * self.f(x + self.DELTA, y + k3)

        return y + (k1 + 2 * k2 + 2 * k3 + k4) / 6

    x = [i for i in np.arange(x0, x + self.DELTA, self.DELTA)]
    y = [y0]

    for i, v in enumerate(x):
        # first one is just an IVP
        if i == 0:
            continue

        # according to the formula
        value = calculate(x[i - 1], y[i - 1])

        y.insert(i, value)

    # just to make sure that dimensions are the same
    x, y = x[:len(y)], y[:len(x)]

    return x, y
```

Calculating errors

Local error

Local error is just a difference between *refrence solution* and some method.

In my implementation I have class-level vars for caching results of each method, therefore for calculating errors of specific method code looks like this:

```
def calculate_euler_errors(self):
       errors, max = self.extract_errors(self.reference_calculated, self.euler_calc
       return errors, max
```

Where `extract_errors()` looks like:

```
def extract_errors(self, reference, calculated):
    errors = []
    max = 0

    for i in range(len(reference) - 1):
        # if our step is bigger than delta => we had an asymptote
        if (i > 0 and abs(reference[i] - reference[i - 1]) < self.DELTA) or i == 0:
            value = abs(reference[i] - calculated[i])
            if value != float('inf'):  # let's skip errors which is inf
                errors.append(value)
                if max < value:
                    max = value

    return errors, max
```

First value is array of differencies on each step, second — max error which will be used in calculating global error

Global error

Global error is maximum local error for given step. For showing dependence from interval I implemented such function:

```
def global_errors(start, end, step):
    """Calculates global error in given range"""

    ...

    euler_errors = []
    improved_euler_errors = []
    runge_kutta_errors = []

    r = range(int(start), int(end), int(step))

    with click.progressbar(r) as bar:
        for i in bar:
            solver = ODESolver(n=i)
            solver.calculate_reference()

            solver.calculate_euler() # calculate points
            err, max = solver.calculate_euler_errors() # calculate errors
            euler_errors.append(max) # save them

            solver.calculate_improved_euler()
            err, max = solver.calculate_improved_euler_errors()
            improved_euler_errors.append(max)

            solver.calculate_runge_kutta()
            err, max = solver.calculate_runge_kutta_errors()
            runge_kutta_errors.append(max)

    ...
```

Code part of implementation

Initialization

All durty stuff are hidden inside `ODESolver` class. Contructior of this class is pretty hacky because of existing asymptote. As a user you probably won't know this hack.

```
def __init__(self, initial_x, initial_y, ending_x, n=1000):
    # if our initial x is on right-side from our asymptote, we will use only one seg
    if initial_x > self.ASYMPTOTE:
        self.INITIAL_X_1 = 0
        self.INITIAL_Y_1 = 0
        self.ENDING_X_1 = 0

        self.INITIAL_X_2 = initial_x
        self.INITIAL_Y_2 = initial_y
        self.ENDING_X_2 = ending_x
        self.DELTA = abs((ending_x - initial_x)) / n
    # otherwise we need to split caclulation into 2 segments
    else:
        self.INITIAL_X_1 = initial_x
        self.INITIAL_Y_1 = initial_y
        self.ENDING_X_1 = self.ASYMPTOTE

        self.INITIAL_X_2 = 1.4  # precalculated and hardcoded IVP after asymptote
        self.INITIAL_Y_2 = -21.76698207998232
        self.ENDING_X_2 = ending_x
        self.DELTA = (abs((self.ASYMPTOTE - initial_x)) + abs((self.ENDING_X_2 - se
```

Calculations

All calculations handle by specific methods `calculate_*method_name*` . One of them:

```
def calculate_reference(self):
    x1, y1 = [], []
    # we need this if for handling case with one segment
    if self.ENDING_X_1 != self.INITIAL_X_1:
        x1, y1 = self.reference_solution(self.INITIAL_X_1, self.INITIAL_Y_1, self.EN
    x2, y2 = self.reference_solution(self.INITIAL_X_2, self.INITIAL_Y_2, self.ENDIN

    # merging both segments into one
    reference_x = x1 + x2
    reference_y = y1 + y2

    self.reference_calculated = reference_y

    return reference_x, reference_y
```

CLI

For user convenience I supposed to use CLI wrapper for the class. It uses click (https://click.palletsprojects.com/en/7.x/) library and has 2 commands:

```
Alexey-top:programming1 de_user$ python main.py --help
Usage: main.py [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  global-errors  Calculates global error in given range
  plot-graphs    Command for plotting graphs
```

`plot-graphs` command:

```
Alexey-top:programming1 de_user$ python main.py plot-graphs --help
Usage: main.py plot-graphs [OPTIONS]

  Command for plotting graphs

Options:
  -x0 FLOAT   X coordinate for IVP
  -y0 FLOAT   Y coordinate for IVP
  -x FLOAT  Ending X coordinate
  -n INTEGER  Number of grid steps
  --help      Show this message and exit.
```

`global-errors` command:

```
Alexey-top:programming1 de_user$ python main.py global-errors --help
Usage: main.py global-errors [OPTIONS]

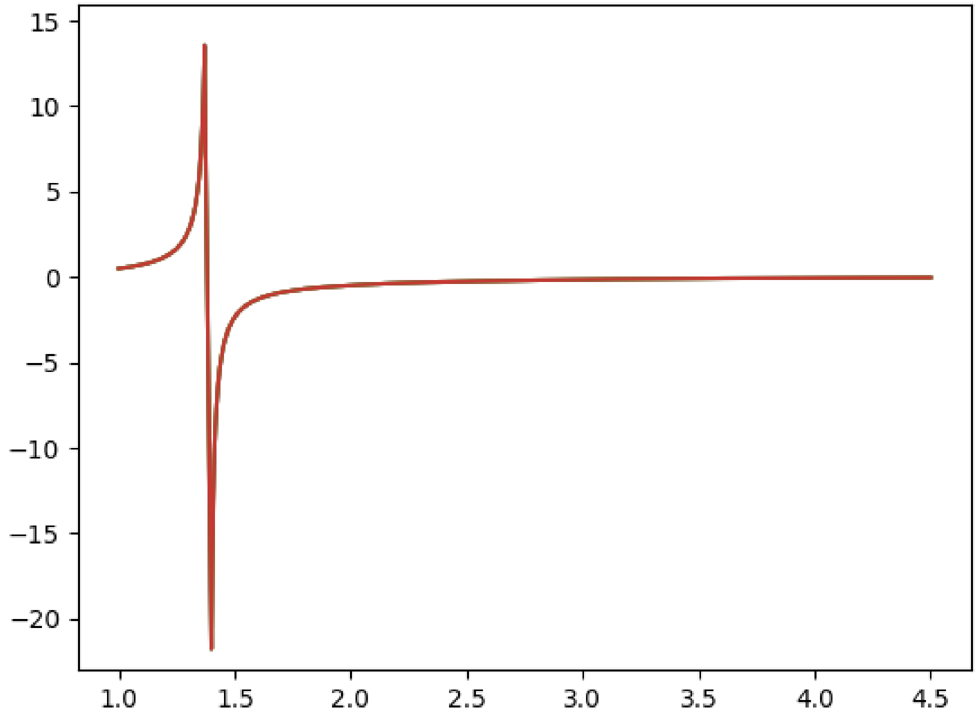
  Calculates global error in given range

Options:
  --start INTEGER  Starting number of grid steps
  --end INTEGER    Ending number of grid steps
  --step INTEGER   Size of each step
  --help           Show this message and exit.
```

Examples

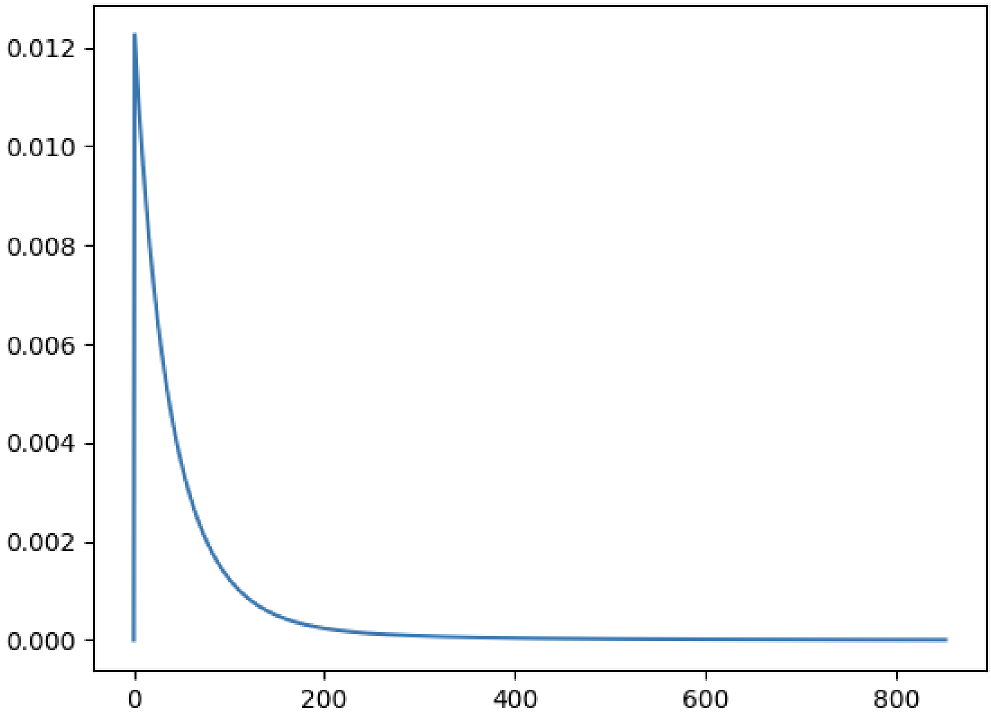
plot-graphs

1. `python main.py plot-graphs -x0 1 -y0 0.5 -x 4.5 -n 10000`

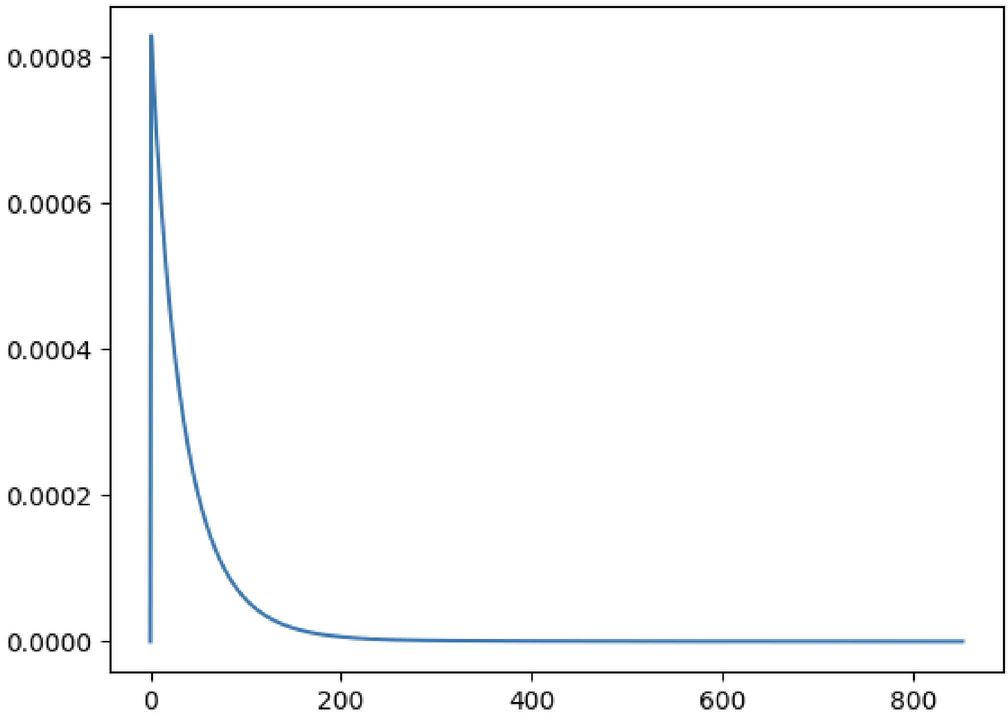


Local errors:

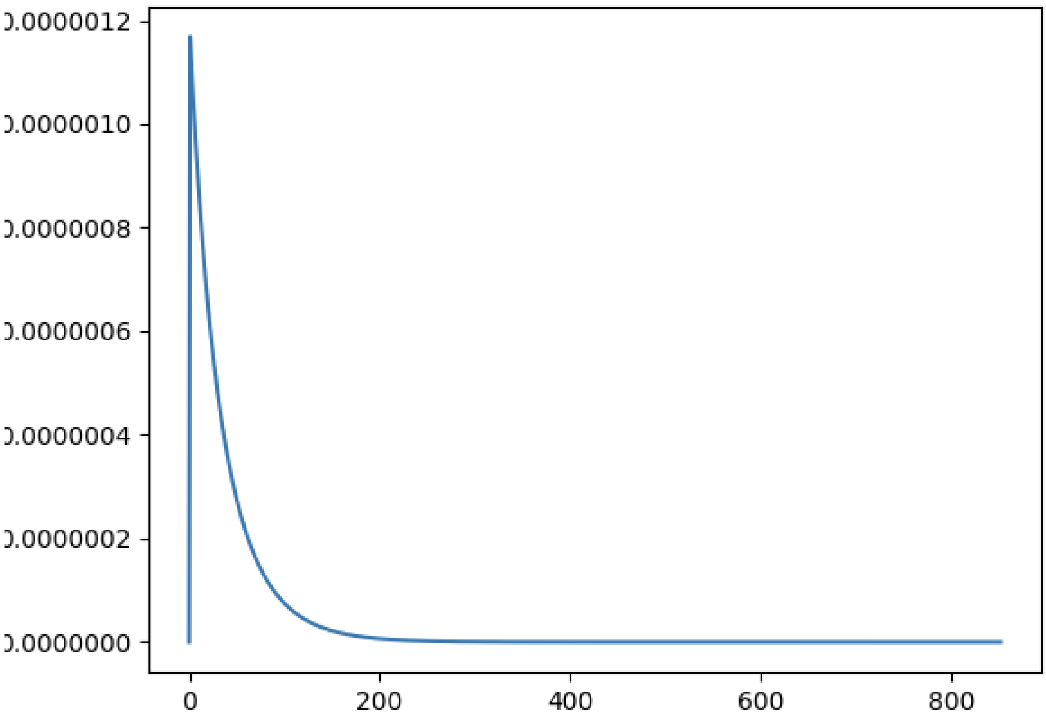
- `Euler`



- Improved Euler

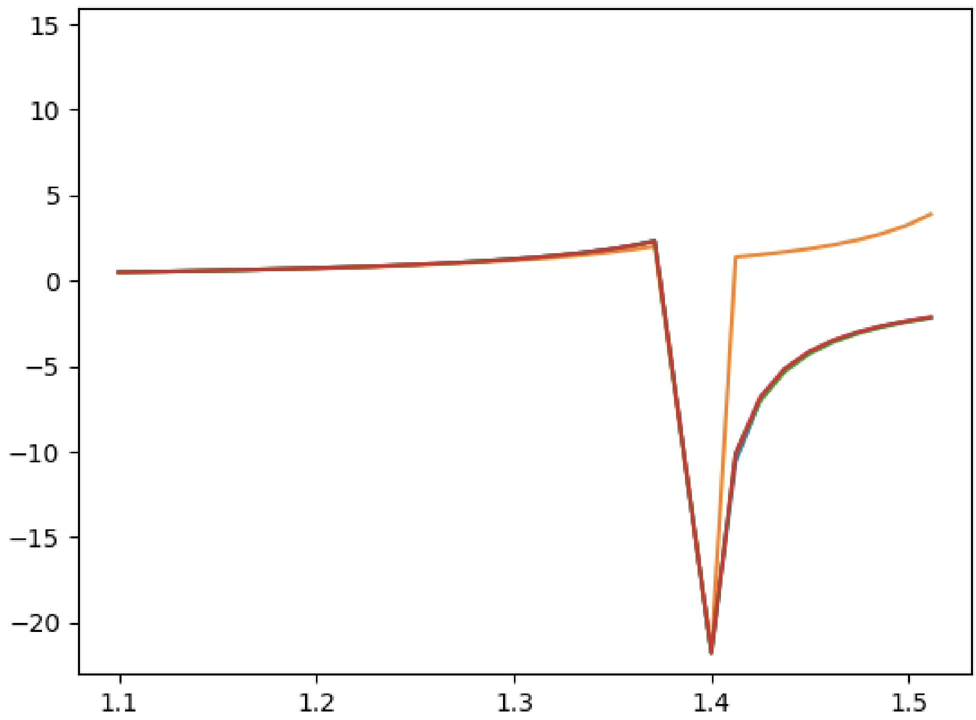


- Runge-Kutta



Note: in this particular equation on some intervals error tends to zero. Therefore sometimes graph of local errors will be empty

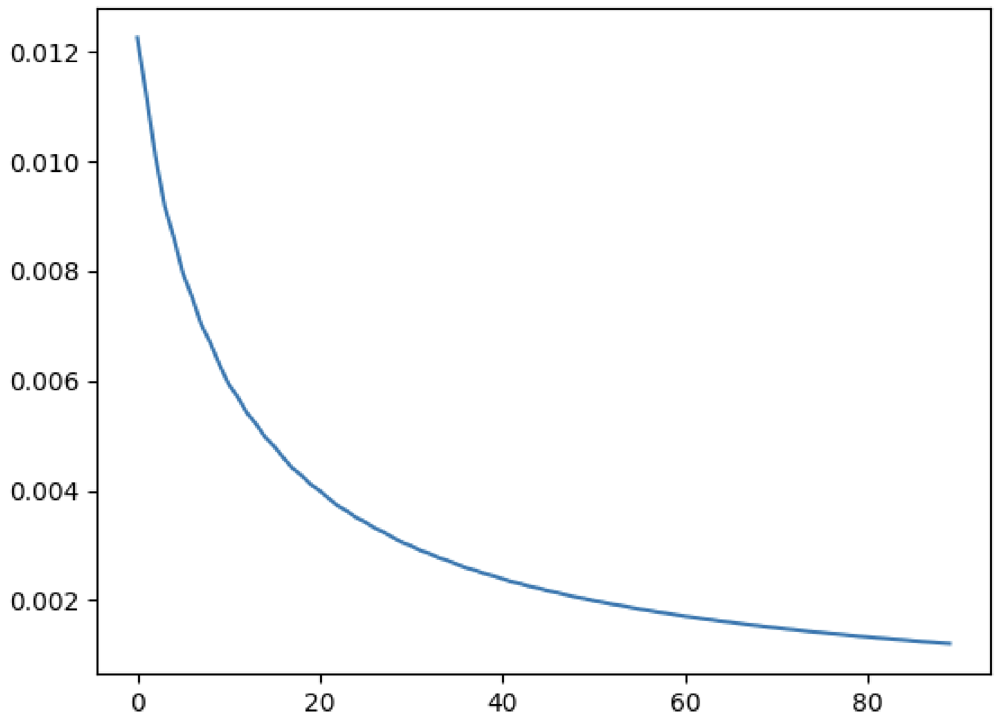
1. `python main.py plot-graphs -x0 1.1 -y0 0.5 -x 1.5 -n 30`



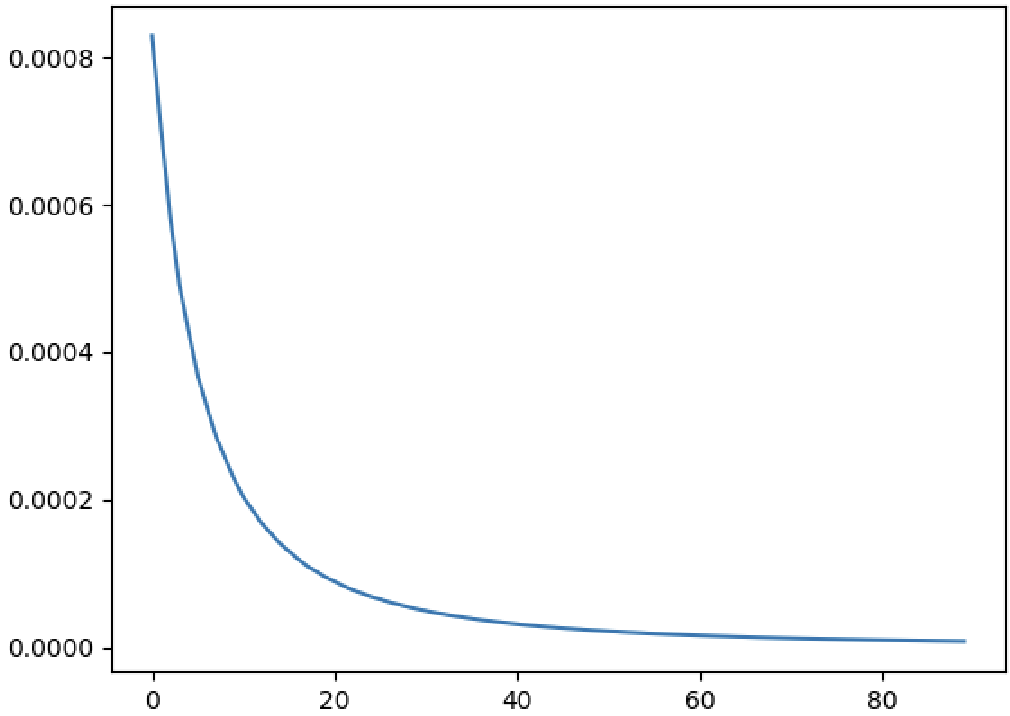
global-errors

1.
```
python main.py global-errors --start=1000 --end=10000 --step=100
Calculating global errors for segment [1000;10000] with step 100.
  [####################################]  100%
```

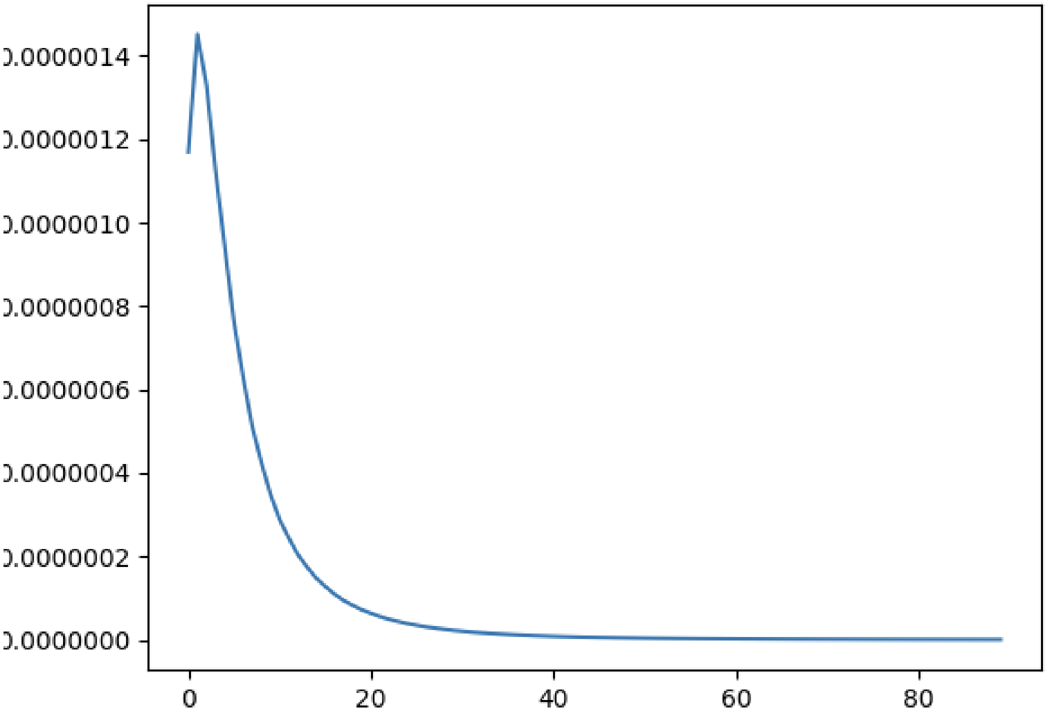
- Euler



- Improved Euler



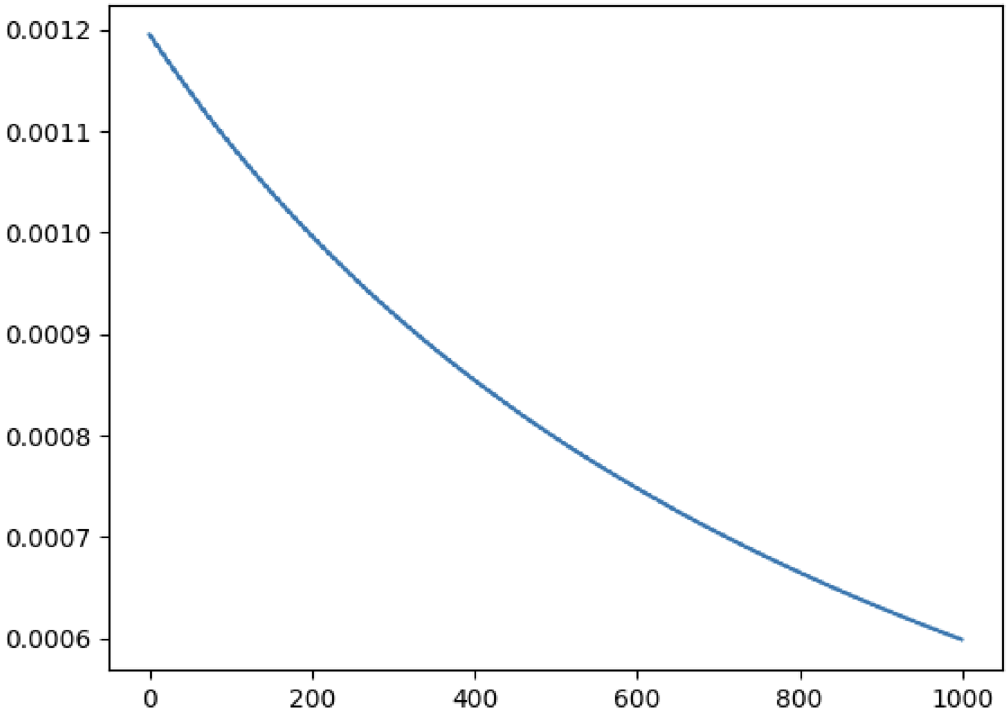
- Runge-Kutta .



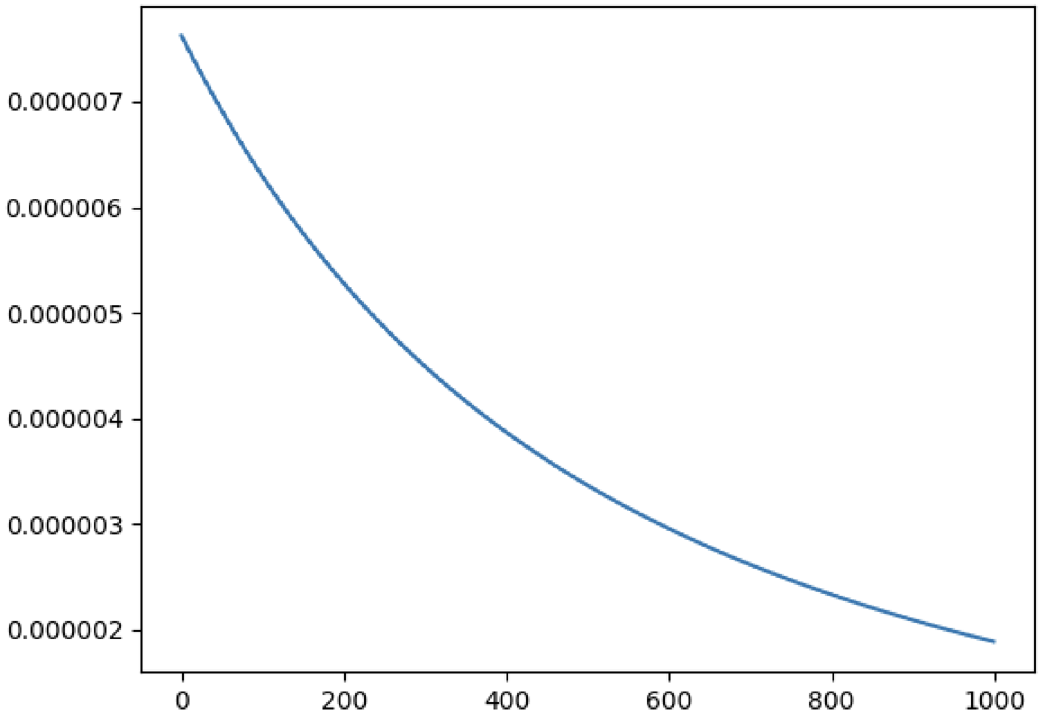
Note: Probably this anomaly happend because of floating point calculations.

2. ```
   python main.py global-errors --start=10000 --end=20000 --step=10
   Calculating global errors for segment [10000;20000] with step 10.
     [####################################]  100%
   ```

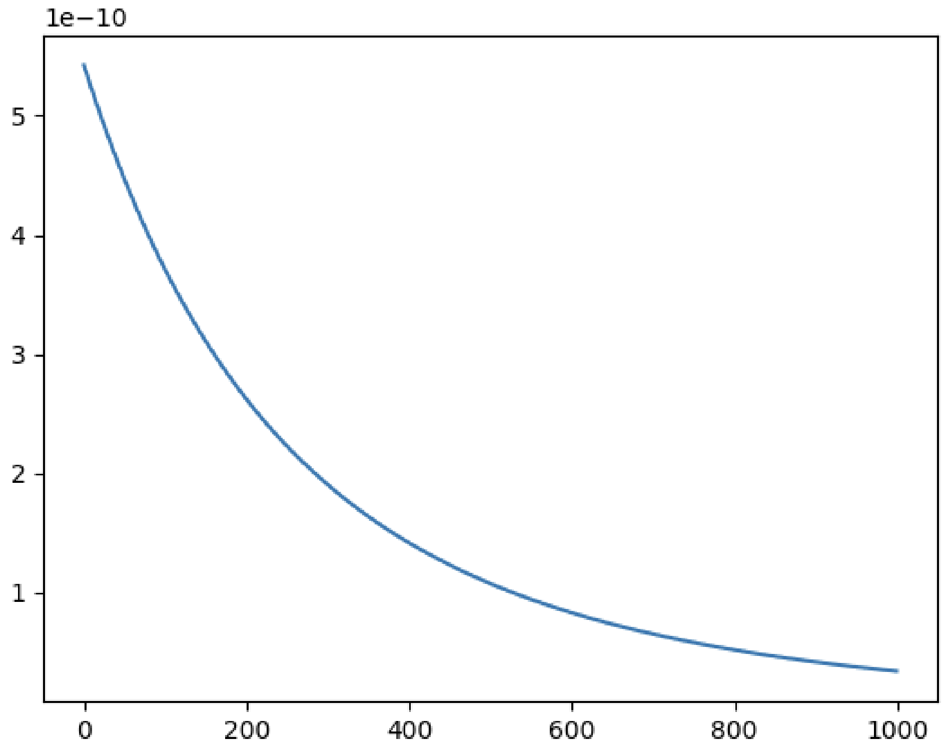
- Euler


   ```

- `Improved Euler`



- `Runge-Kutta`



Web-interface

There is also an alpha-version web-interface. You can try it running `app.py` and accessing `localhost:8080`. **Careful, it may crush your browser!** (better to use FireFox).