Danil Kabirov B17-03

Report

Differential Equation Graph builder

Variant 5:

y’ = cos(x) - y

Exact solution for equation is:

$$y(x) = c_1\, e^{-x} + \frac{\sin(x)}{2} + \frac{\cos(x)}{2}$$

Solution graphs

Graph builder allow user to draw plots of solution with custom ***x0***, ***y0***, ***N*** and ***Max x*** (right edge of the calculation segment).

Exact solution

To calculate points GUI calculates constant and calculate y-points using exact function:

```
def calculate(self, x0, y0, tox, n, **kwargs):
    if x0 >= tox: return []
    self.const = self.__calculate_const(x0, y0) # const = 0.8402572149116814
    points = []
    h = (tox - x0) / n
    x = x0
    while x <= tox:
        points.append((x, self.__func(x)))
        x += h
    return points

def __calculate_const(self, x0, y0):
    return -(-y0 + math.sin(x0) / 2 + math.cos(x0) / 2) / math.exp(-x0)

def __func(self, x):
    return self.const * math.exp(-x) + math.sin(x) / 2 + math.cos(x) / 2
```

Euler method

Euler method multiplies ***h*** on ***y'(x)*** to calculate ***yi*** for each iteration*:*

```
class Euler(CalculationMethod):
    def calculate(self, x0, y0, tox, n, **kwargs):
        if x0 >= tox: return []
        h = (tox - x0) / n
        points = []
        x = x0
        y = y0
        while x <= tox:
            points.append((x, y))
            y = y + h * self.__func(x, y)
            x += h

        return points

    def __func(self, x, y):
        return math.cos(x) - y
```

Improved Euler method:

Improved Euler method increases the accuracy of the approximation by calculating the average derivative between the current and the next point:

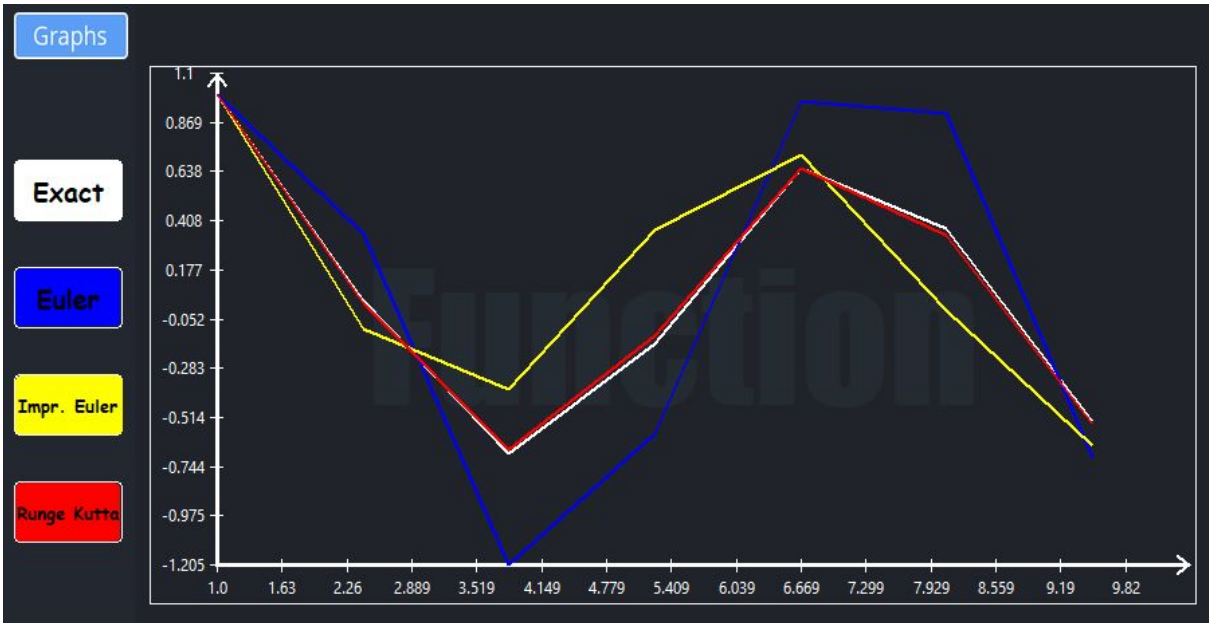
```
class ImprovedEuler(CalculationMethod):
    def calculate(self, x0, y0, tox, n, **kwargs):
        if x0 >= tox: return []
        h = (tox - x0) / n
        points = []
        x = x0
        y = y0
        while x <= tox:
            points.append((x, y))
            prev_y = y
            y = y + h * self.__func(x, y)
            y = prev_y + h * (self.__func(x, prev_y) + self.__func(x + h, y)) / 2
            x += h

        return points

    def __func(self, x, y):
        return math.cos(x) - y
```

Runge-Kutta method:

Runge-Kutta method using the same approach but has 4 steps, except 2:

```python
class RungeKutta(CalculationMethod):
    def calculate(self, x0, y0, tox, n, **kwargs):
        if x0 >= tox: return []
        h = (tox - x0) / n
        points = []
        x = x0
        y = y0
        while x <= tox:
            points.append((x, y))
            k1 = self.__func(x, y)
            k2 = self.__func(x + h/2, y + h*k1/2)
            k3 = self.__func(x + h/2, y + h*k2/2)
            k4 = self.__func(x + h, y + h * k3)
            y = y + (k1 + 2 * k2 + 2 * k3 + k4) * h / 6
            x = x + h

        return points

    def __func(self, x, y):
        return math.cos(x) - y
```

The graph allows to compare the results of the methods:



Graphs was builded with initial values:

x0 = 1, y0 = 1, max X = 9.5, N = 6 (So, every graph has 7 points)

As we can see, Runge-Kutta method has best results.

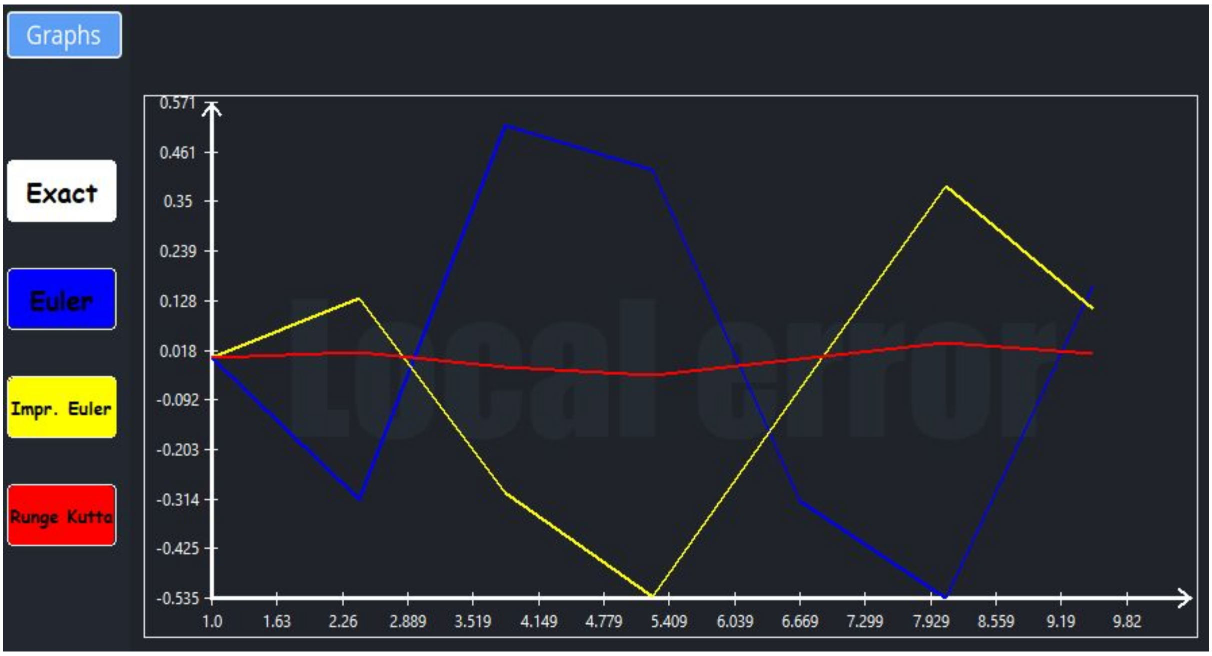
Local error graphs

To calculate Local error of Euler method, algorithm subtracts calculated value (***yi***) from exact solution value (***y(xi)***):

```
class EulerLocalError(CalculationMethod):
    def calculate(self, x0, y0, tox, n, **kwargs):
        if x0 >= tox: return []
        h = (tox - x0) / n
        points = []
        x = x0
        y = y0
        i = 0
        while x <= tox:
            points.append((x, kwargs['exact_solution'][i][1] - y))
            y = y + h * self.__func(x, y)
            x += h
            i += 1

        return points

    def __func(self, x, y):
        return math.cos(x) - y
```

The same line was added in each of function to calculate local error of other methods.

Now we can compare the results:



Graphs was builded with the same settings.

Algorithm takes non-absolute values, that is why it looks like trigonometric functions.

Thanks to non-absolute values, we can observe in which points graphs intersects with exact solution(and with each other) by finding the points of intersection with the x-axis.

As we can see, Runge-Kutta method local error graph is closest to 0, Euler is outermost it. So, Runge Kutta has the best results, Euler - the worst.

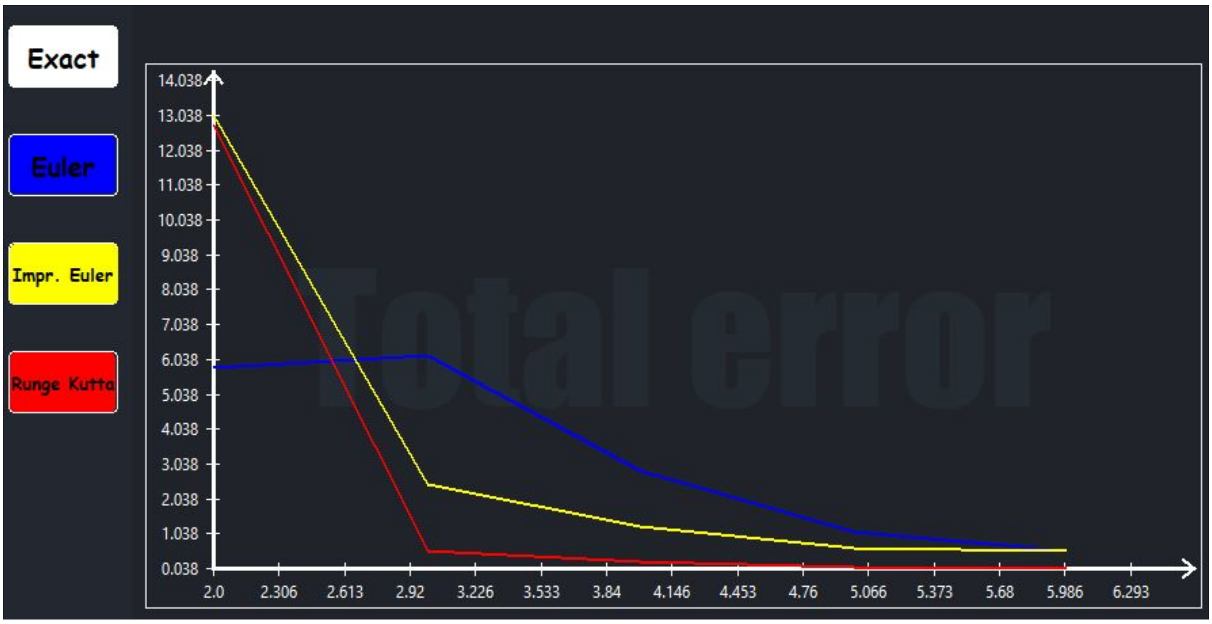
Global error graphs

To calculate global error, algorithm(for every method) takes exact solution for current ***N*** (number of parts to divide the initial segment) and local error of this solution. Finds the maximum absolute value of local error and continues to do the same for other ***N****s*

```
class EulerGlobalError(CalculationMethod):
    def calculate(self, x0, y0, tox, n, **kwargs):
        if x0 >= tox: return []
        h = (tox - x0) / n
        points = []

        div = kwargs['min_division']
        while div <= kwargs['max_division']:
            exact_solution_points = ExactSolution().calculate(x0, y0, tox, div)
            local_error_points = EulerLocalError().calculate(x0, y0, tox, div, exact_solution=exact_solution_points)
            max_local_error = 0
            for pnt in local_error_points:
                max_local_error = max(max_local_error, abs(pnt[1]))

            points.append((div, max_local_error))

            div += 1

        return points
```

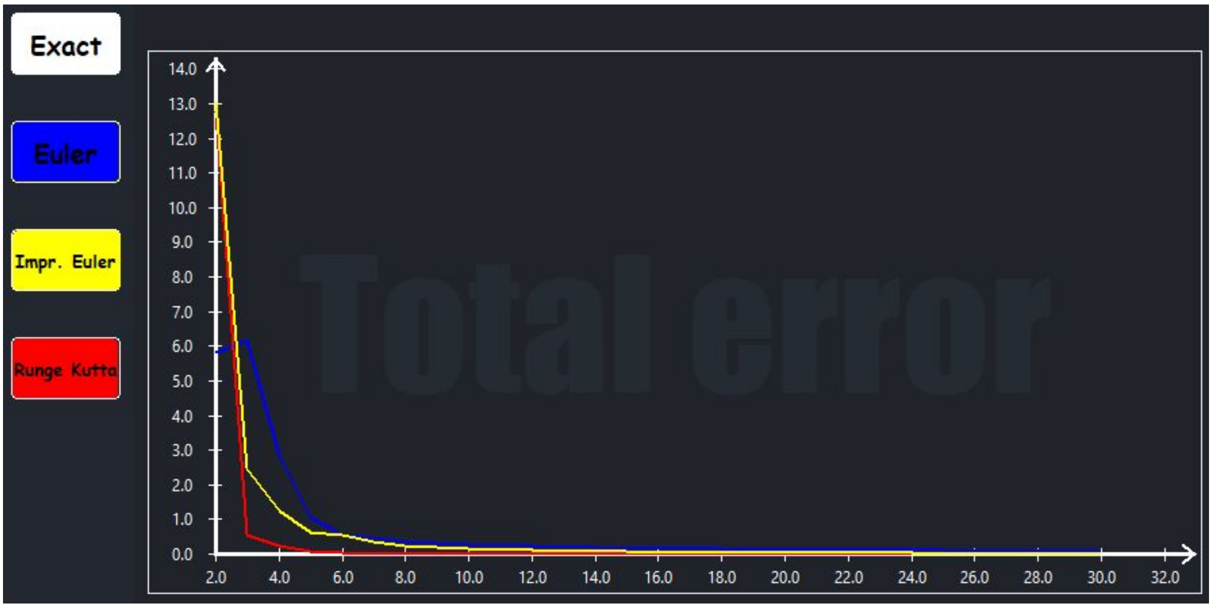
Code for other methods is similar

Now we can consider graphs (*Graphs was builded on segment [2, 6]*):



Runge-Kutta-graph and Improved Euler-graph is bigger than Euler one in the beginning. It can be explained by the specificity of the exact solution graph:
In the beginning out graph is divided to 2 parts. It means that every graph has 3 points: x0=0, x1=(9.5-1)/2 = 4.25 and x2 = 9.5. On every segment (x0->x1 and x1->x2) exact solution graph does not have a strong tendency to increase or decrease. It changes in different sides. So, it's almost impossible to guess the right point with high ***h*** (grid step).
The graph of euler global error is closer than other on ***N***=2 due to chance.

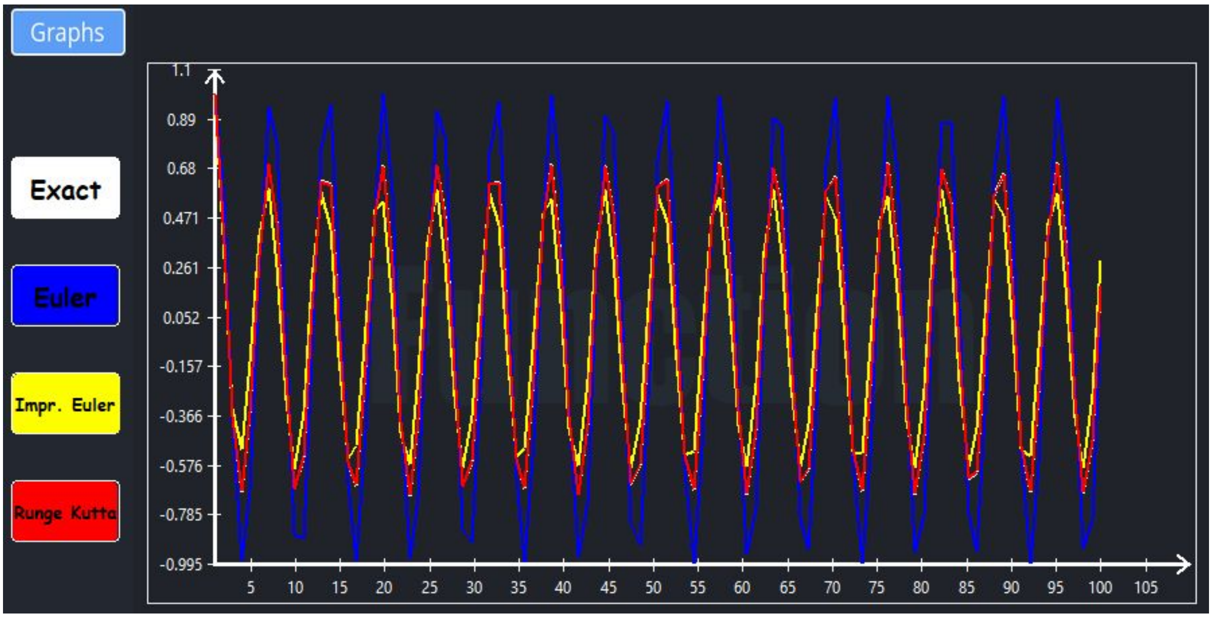
We also can build graphs of global errors for maximum ***N***=30



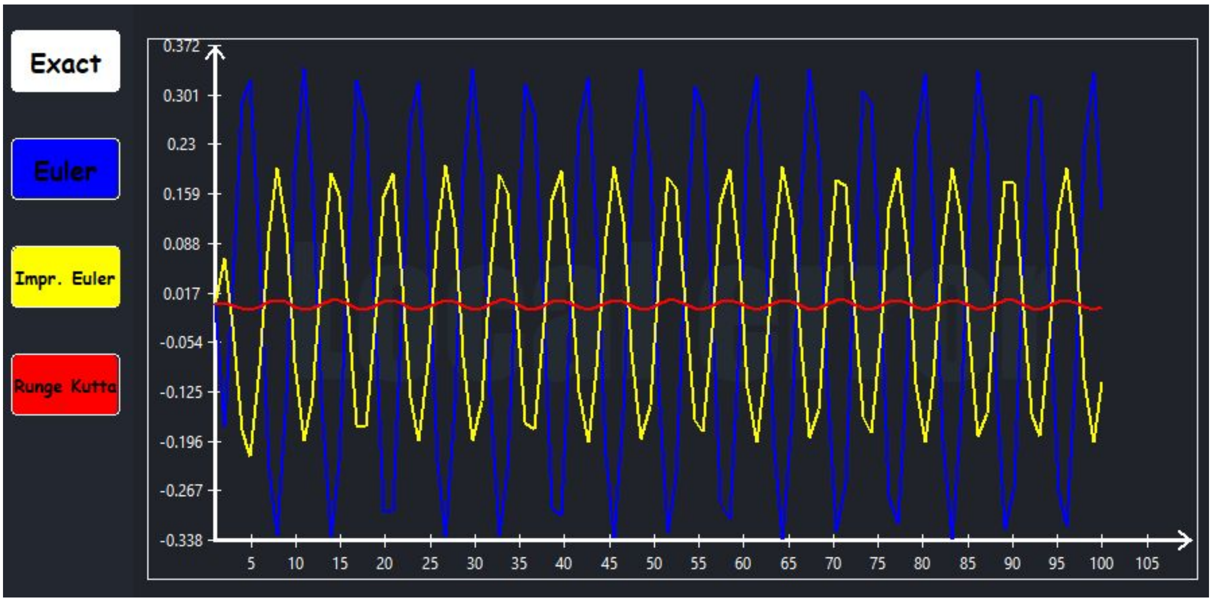
Graphs was builded on segment [2, 30]

Now we can make sure that Runge-Kutta-method has the lowest total error with not small ***N.*** Euler-method has the greatest one.

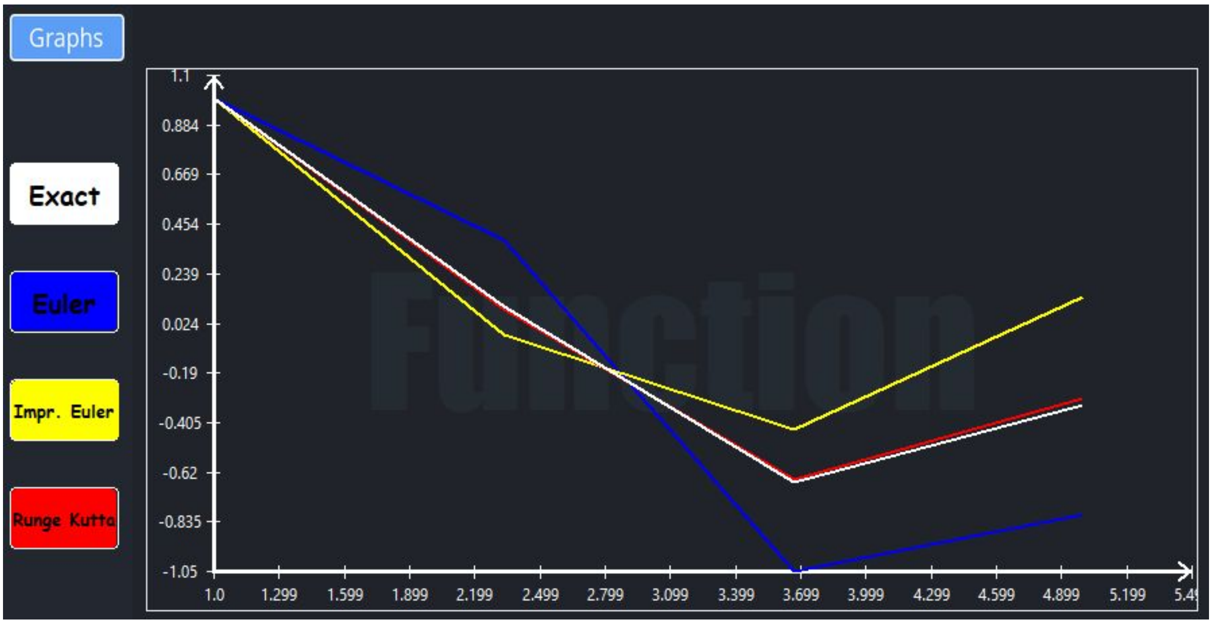
More graphs:



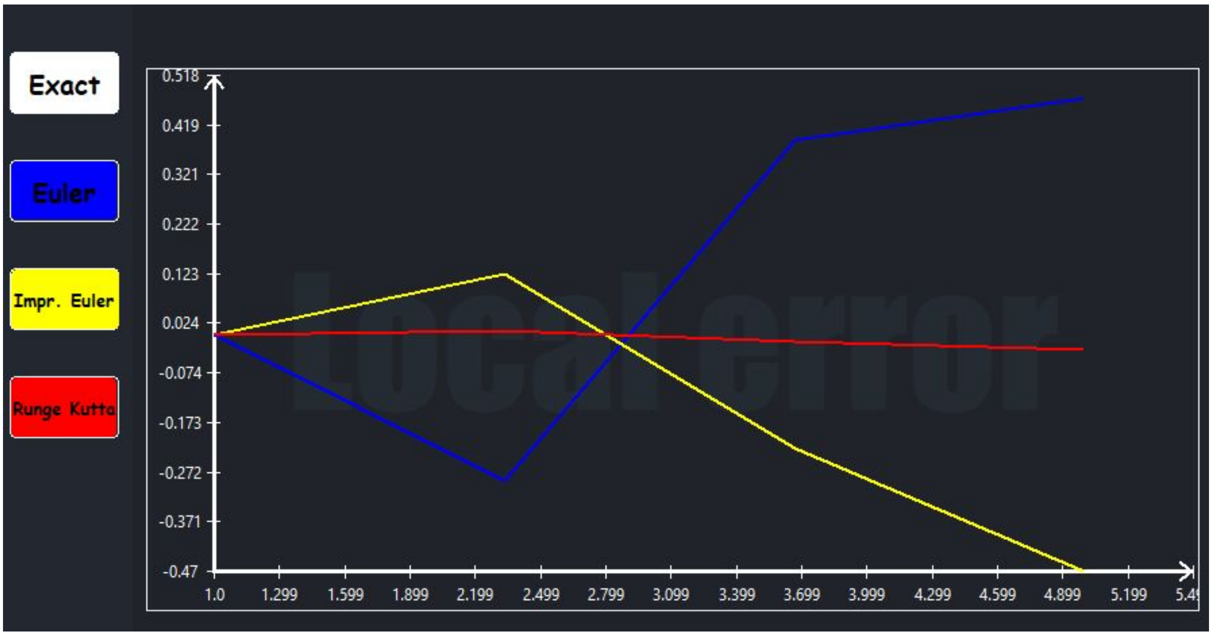
Function Graph. *Graph was builded with initial values:*
x0 = 1, y0 = 1, max X = 100, N = 100



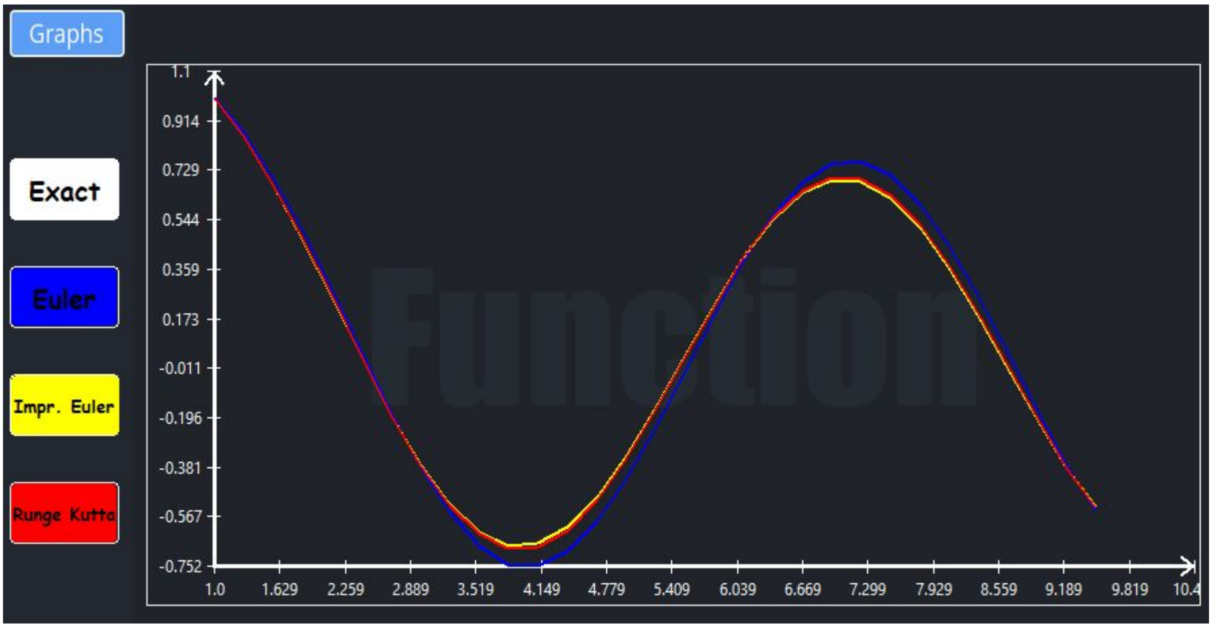
Local Error Graph. *Graph was builded with initial values:*
x0 = 1, y0 = 1, max X = 100, N = 100



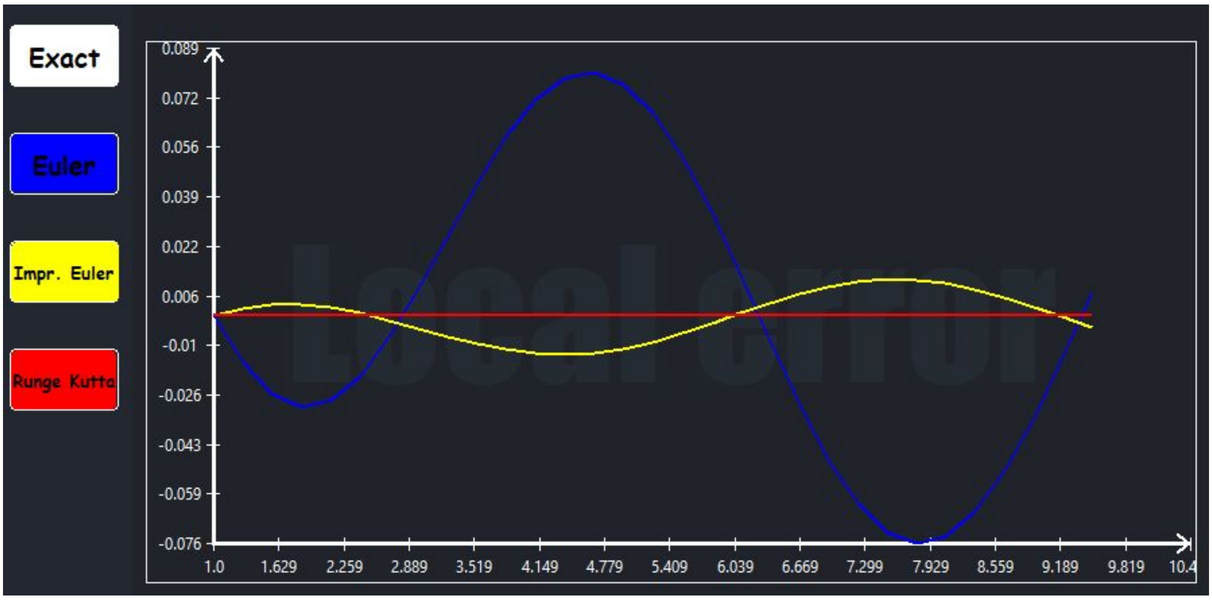
Function Graph. *Graph was builded with initial values:*
x0 = 1, y0 = 1, max X = 5, N = 3



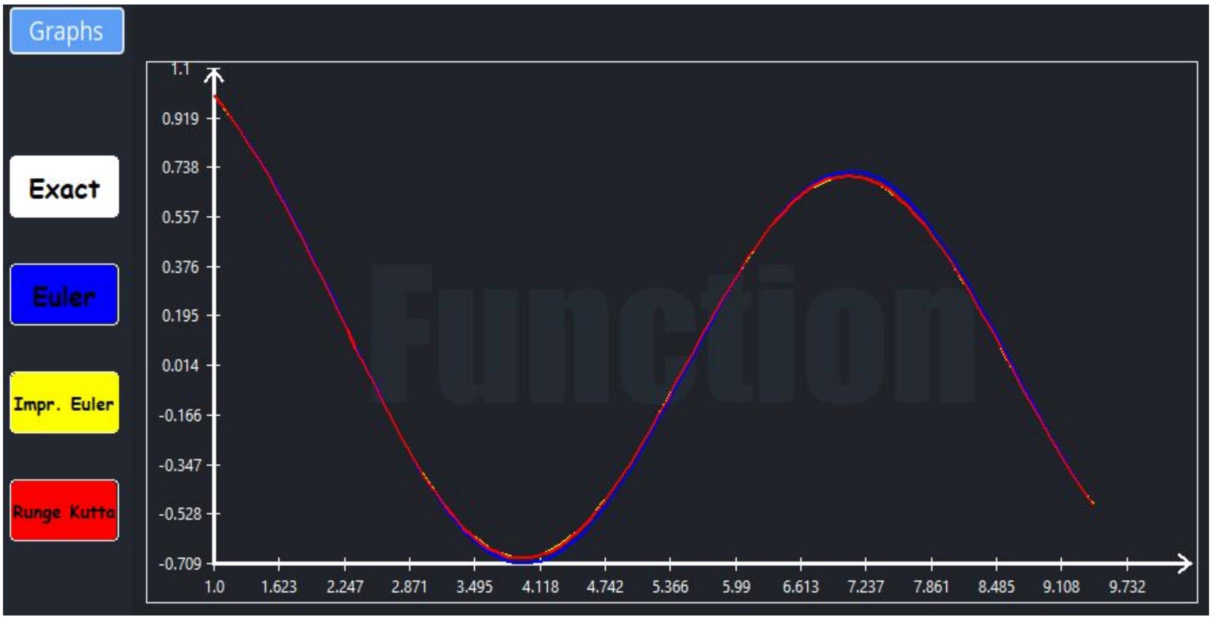
Local Error Graph. *Graph was builded with initial values:*
x0 = 1, y0 = 1, max X = 5, N = 3



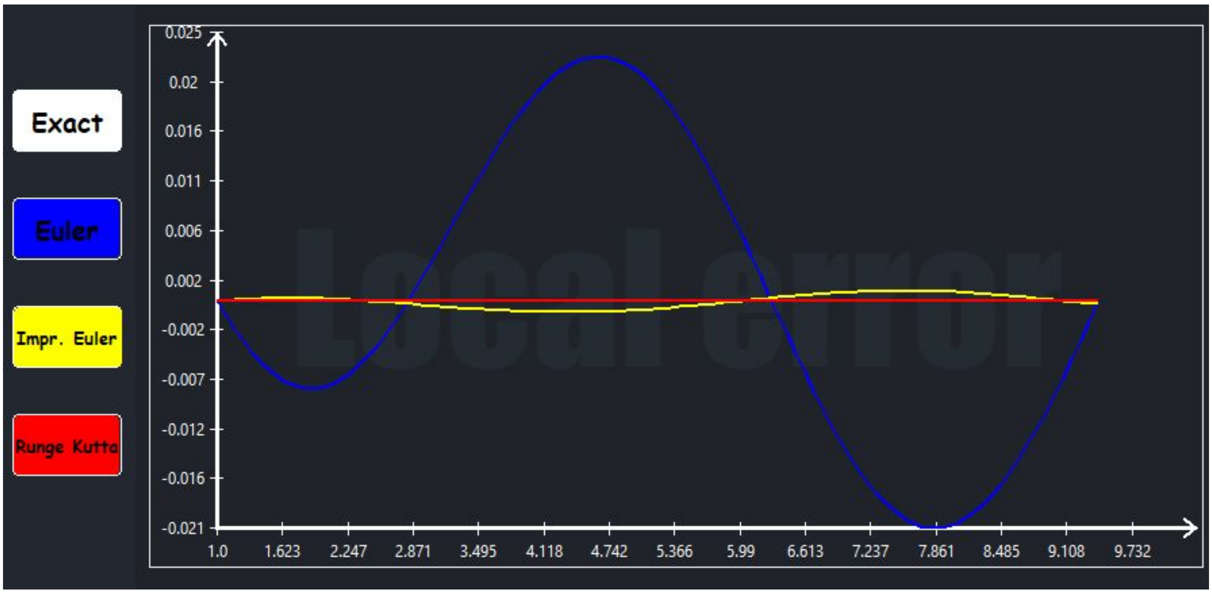
Function Graph. *Graph was builded with initial values:*
x0 = 1, y0 = 1, max X = 9.5, N = 30



Local Error Graph. *Graph was builded with initial values:*
x0 = 1, y0 = 1, max X = 9.5, N = 30



Function Graph. *Graph was builded with initial values:*
x0 = 1, y0 = 1, max X = 9.5, N = 100



Local Error Graph. *Graph was builded with initial values:*
x0 = 1, y0 = 1, max X = 9.5, N = 100