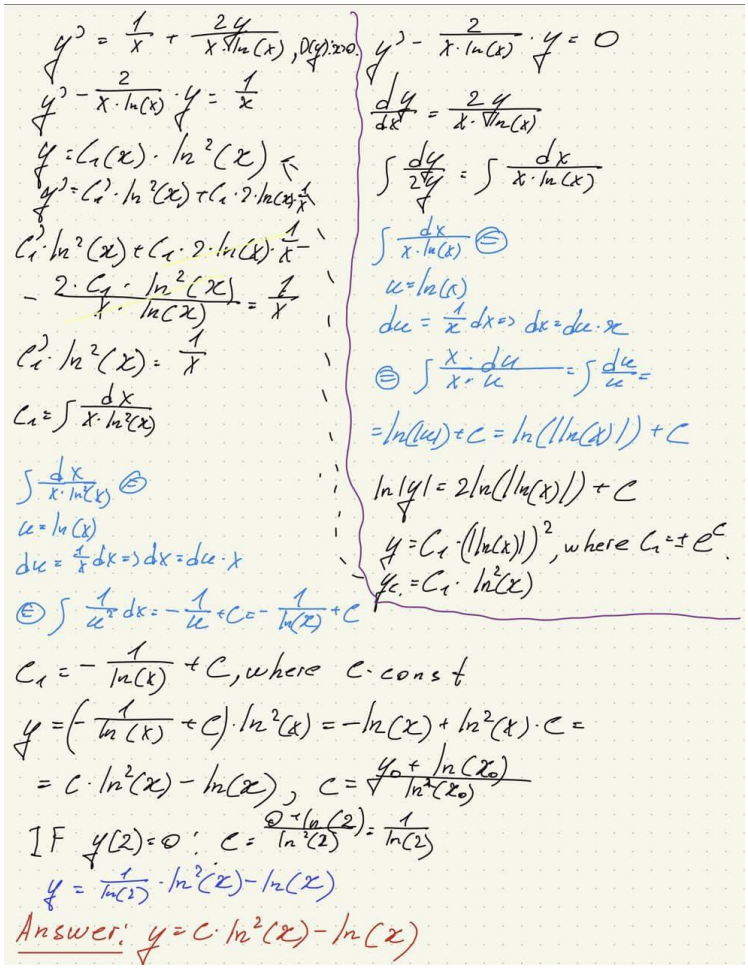
Report

Link to github: https://github.com/Kirill-Kuznetsov-git/DiffEq-Assignment

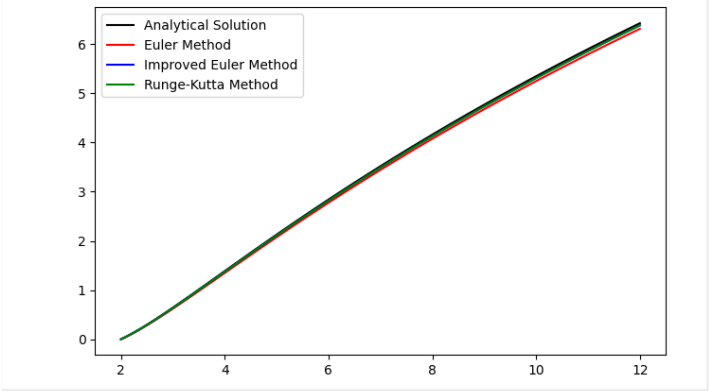
Solution:



Screenshots of results:

Differential Equations

Solutions | LTE | GTE



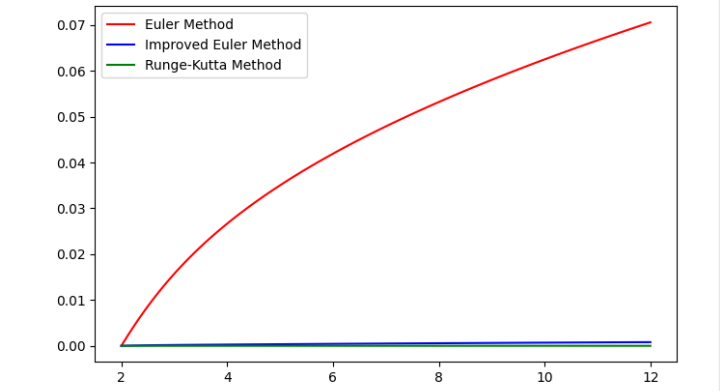
Analytical Solution
Euler Method
Euler Improved Method
Runge-Kutta Method

x_0: 2
y_0: 0
X: 12
N: 120
n_0: 10

1/x+2*y/(x*ln(x))

Differential Equations

Solutions | LTE | GTE



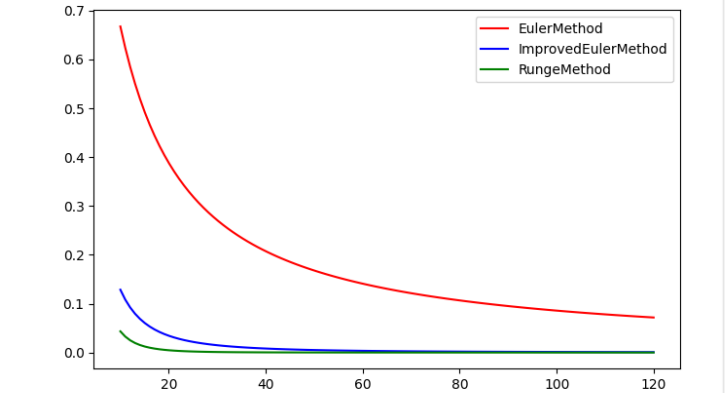
Analytical Solution
Euler Method
Euler Improved Method
Runge-Kutta Method

x_0: 2
y_0: 0
X: 12
N: 120
n_0: 10

1/x+2*y/(x*ln(x))

Differential Equations

Solutions | LTE | GTE

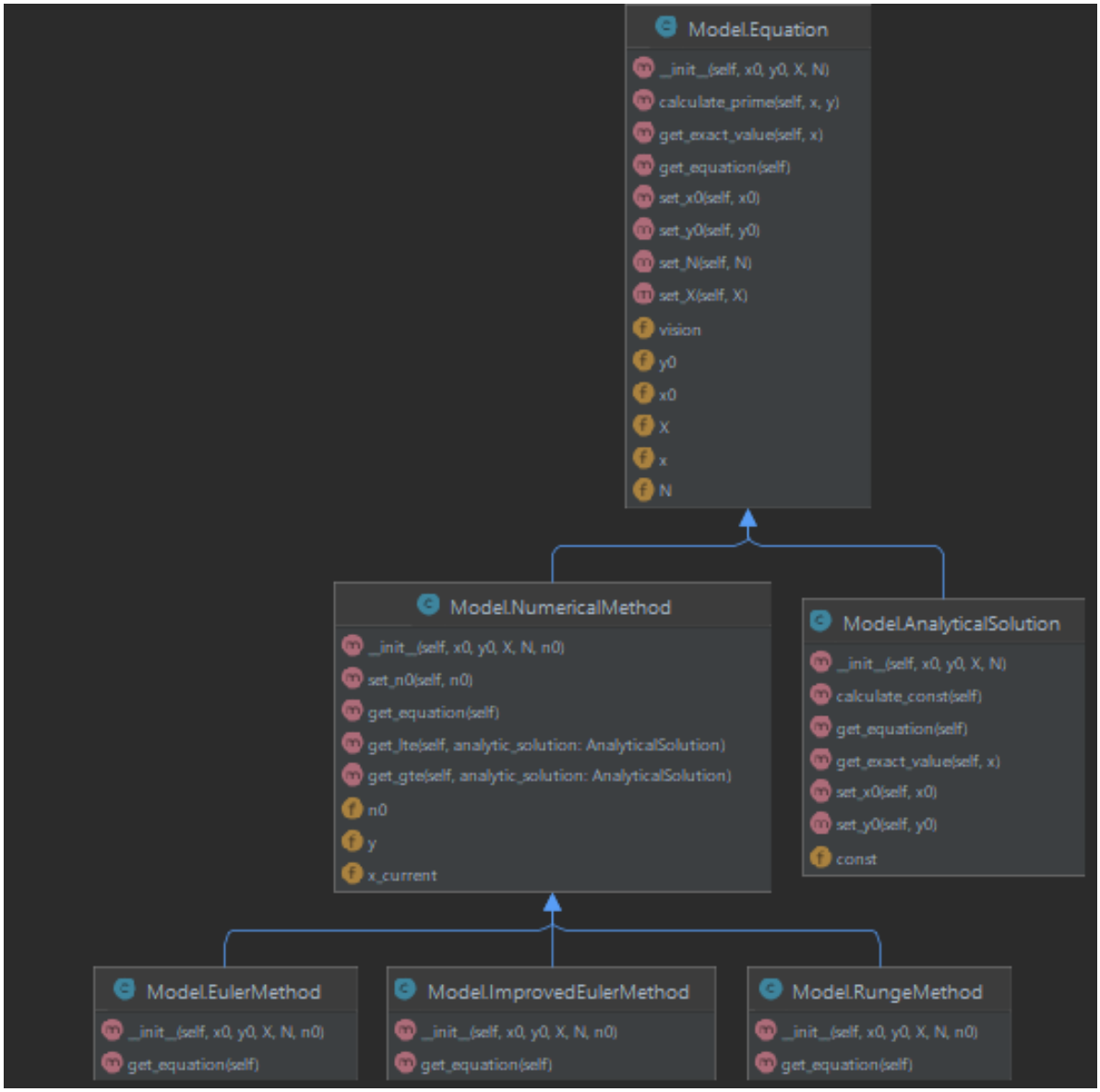


Analytical Solution
Euler Method
Euler Improved Method
Runge-Kutta Method

x_0: 2
y_0: 0
X: 12
N: 120
n_0: 10

1/x+2*y/(x*ln(x))

UML diagram:



CODE:

I my model, I have abstract Equation model.

It model has some set value method.

Function “calculate prime” - return a value of f(x, y).

Function “get_equation()” return a equation in a form of np.linespace and python array.

```python
class Equation:
    def __init__(self, x0, y0, X, N):
        self.x0 = x0
        self.X = X
        self.y0 = y0
        self.N = N
        self.x = np.linspace(self.x0, self.X, self.N)
        self.vision = True

    def calculate_prime(self, x, y):
        return 1 / x + 2 * y / (x * np.log(x))

    def get_exact_value(self, x):
        pass

    def get_equation(self):
        pass

    def set_x0(self, x0):
        self.x0 = x0
```

From Equation extended two other classes:

1) Numerical Method.(Abstract class wich represent all numerical methods.)

Fucntions “get_lte()” and “get_gte()” return arrays of lte and gte respectivly.

```python
def get_lte(self, analytic_solution: AnalyticalSolution) -> list:
    errors = []
    h = (self.X - self.x0) / self.N
    self.y = self.get_equation()[1]
    self.x_current = self.x0
    for i in range(self.N):
        errors.append(abs(self.y[i] - analytic_solution.get_exact_value(self.x_current)))

        self.x_current += h
    return [analytic_solution.x, errors]

def get_gte(self, analytic_solution: AnalyticalSolution):
    if self.n0 >= self.N:
        return [[0], [0]]
    errors = []
    n_old = self.N
    for i in range(int(self.N - self.n0)):
        self.set_N(self.n0 + i - 1)
        errors.append(max(self.get_lte(analytic_solution)[1]))
    self.set_N(n_old)
    return [np.linspace(self.n0, self.N, int(self.N - self.n0)), errors]
```

2) Analytic Method.

```
class AnalyticalSolution(Equation):
    def __init__(self, x0, y0, X, N):
        super().__init__(x0, y0, X, N)
        self.const = (self.y0 + np.log(self.x0)) / (np.log(self.x0) ** 2)

    def calculate_const(self):
        self.const = (self.y0 + np.log(self.x0)) / (np.log(self.x0) ** 2)
        return self.const

    def get_equation(self):
        super().get_equation()
        y = self.const * (np.log(self.x) ** 2) - np.log(self.x)
        return [self.x, y]

    def get_exact_value(self, x):

        return self.const * (np.log(x) ** 2) - np.log(x)

    def set_x0(self, x0):
        super().set_x0(x0)
        self.calculate_const()
```

From a Numerical method extended three other classes:

Each of this classes overlwrite her own get_eqation() fucntion.

1) Euler Method

```
class EulerMethod(NumericalMethod):
    def __init__(self, x0, y0, X, N, n0):
        super().__init__(x0, y0, X, N, n0)

    def get_equation(self):
        res = super().get_equation()
        print(self.y)
        if res: return res
        h = (self.X - self.x0) / self.N
        for i in range(self.N - 1):
            self.y.append(self.y[-1] + h * self.calculate_prime(self.x_current, self.y[-1]))
            self.x_current += (self.X - self.x0) / self.N

        return [self.x, self.y]
```

2) Improved Euler Method0

```
class ImprovedEulerMethod(NumericalMethod):
    def __init__(self, x0, y0, X, N, n0):
        super().__init__(x0, y0, X, N, n0)

    def get_equation(self):
        res = super().get_equation()
        if res: return res
        h = (self.X - self.x0) / self.N
        for i in range(self.N - 1):
            self.y.append(self.y[-1] + h * self.calculate_prime(self.x_current + h / 2,
                                                                self.y[-1] + h / 2 * self.calculate_prime(
                                                                    self.x_current, self.y[-1])))
            self.x_current += (self.X - self.x0) / self.N

        return [self.x, self.y]
```

3) Runge-Katta Method

```
class RungeMethod(NumericalMethod):
    def __init__(self, x0, y0, X, N, n0):
        super().__init__(x0, y0, X, N, n0)

    def get_equation(self):
        res = super().get_equation()
        if res: return res
        h = (self.X - self.x0) / self.N
        for i in range(self.N - 1):
            k1 = self.calculate_prime(self.x_current, self.y[-1])
            k2 = self.calculate_prime(self.x_current + h / 2, self.y[-1] + h / 2 * k1)
            k3 = self.calculate_prime(self.x_current + h / 2, self.y[-1] + h / 2 * k2)
            k4 = self.calculate_prime(self.x_current + h, self.y[-1] + h * k3)
            self.y.append(self.y[-1] + h / 6 * (k1 + 2 * k2 + 2 * k3 + k4))
            self.x_current += (self.X - self.x0) / self.N

        return [self.x, self.y]
```

To visualize graphs, I use a PyQt5 and MatPlotLib.

MainWindow – represent application in general.

FirstWindow, SecondWindow, ThirdWindow – represent each window with gpaphs. With Solutions, LTE and GTE respectivly.

From PyQt5 I use Canvas CheckBoxes and SpinBoxes.