REPORT

Anastassiya Boiko

Group: BS 17-1

Innopolis University

Plan:

1) Exact solution
2) Numerical methods: solutions and errors
3) Program: how it looks
4) Structure of program

Exact solution

Given differential equation $y' = cosx - y$ with default initial values: x_0=1, y_0=1 and right border of X=9.5.

Exact solution of given expression is:

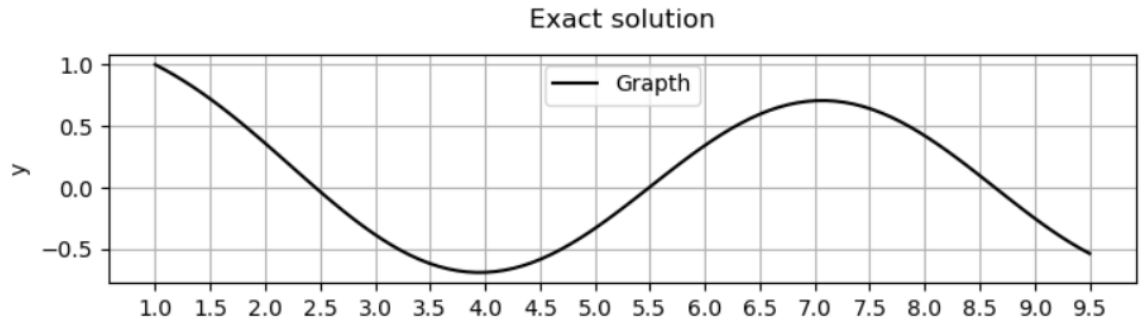
$$y(x) = C_1 * e^{-1} + \frac{\sin x_0}{2} + \frac{cosx_0}{2}$$

After making substitution with given IVP, we get:

$1 = C_1 * e^{-1} + \frac{\sin(1)}{2} + \frac{\cos(1)}{2}$, from which we get $C(x) = 0{,}4336$.

In the application, the graph of exact solution is shown in the second plot.

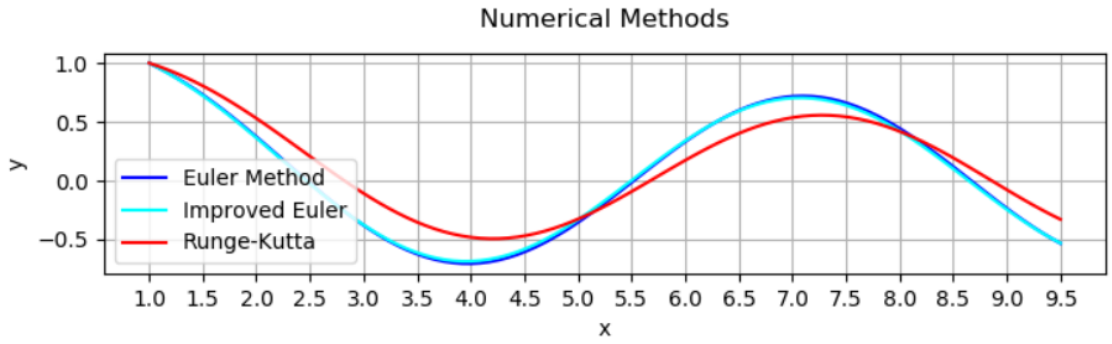
Illustration of graph:



Numerical methods: solutions and errors

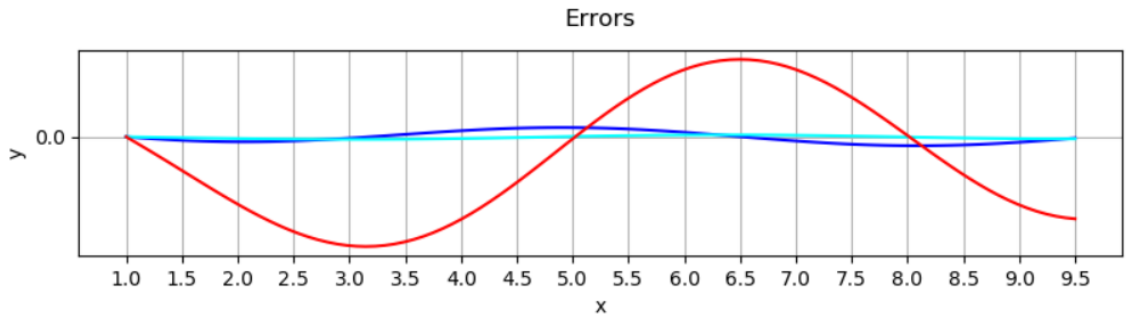
Numerical solutions:

Euler's method, Improved Euler's method and Runge-Kutta method are shown in the first plot. Step depends on the number of points and set by slider.



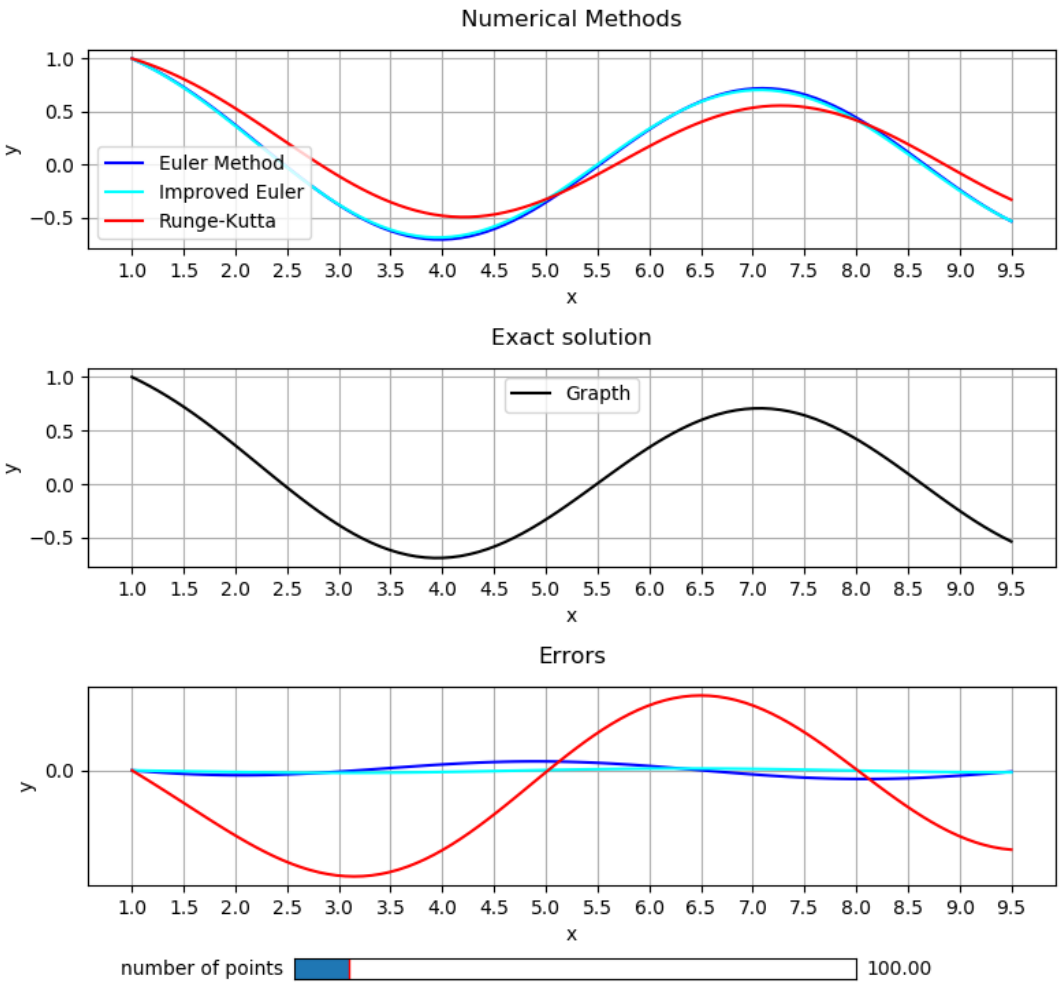
Errors:

We define error graphs by the difference between exact solution and numerical methods.

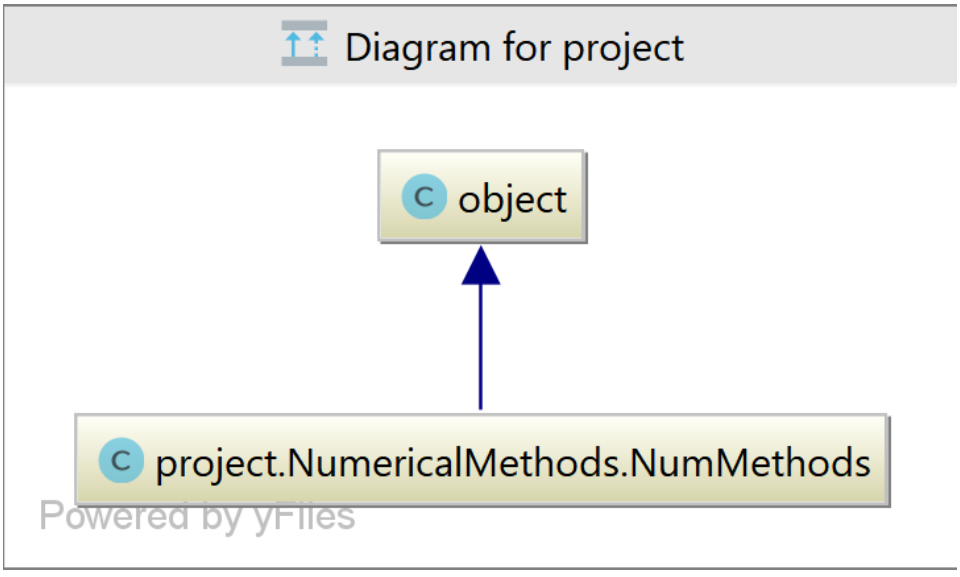


Program: how it looks

Graphical representation:



Structure in diagram:



Structure of the program

Firstly, there is a need to import the class that contains necessary parameters for counting and plotting graphs. Program contains methods which count values for Euler's method, Improved Euler's method, Runge-Kutta, exact solution and errors.

Class initialization:

```
def __init__(self, _xStart, _xFinite, _yStart, _count):
    self.xStart = _xStart
    self.xFinite = _xFinite
    self.yStart = _yStart
    self.count = _count
    self.h = (_xFinite - _xStart) / _count
    self.x = np.linspace(_xStart, _xFinite, _count)
    self.y = np.zeros([_count])
    self.yEuler = np.zeros([_count])
    self.yImprEuler = np.zeros([_count])
    self.yRunge = np.zeros([_count])
    self.errorEuler = np.zeros([_count])
    self.errorImprEuler = np.zeros([_count])
    self.errorRunge = np.zeros([_count])
```

Numerical methods:

```
def eulerMethod(self):
    self.yEuler[0] = self.yStart
    for i in range(1, self.count):
        self.yEuler[i] = self.h * (np.cos(self.x[i - 1]) - self.yEuler[i - 1]) + self.yEuler[i - 1]

def improvedEulerMethod(self):
    self.yImprEuler[0] = self.yStart
    for i in range(1, self.count):
        newX = self.x[i - 1] + self.h / 2
        newY = self.yImprEuler[i - 1] + self.h / 2 * (np.cos(self.x[i - 1]) - self.yImprEuler[i - 1])
        self.yImprEuler[i] = self.h * (np.cos(newX) - newY) + self.yImprEuler[i - 1]

def rungeKutta(self):
    self.yRunge[0] = self.yStart
    for i in range(1, self.count):
        k1 = np.cos(self.x[i - 1]) - self.yRunge[i - 1]
        k2 = np.cos(self.x[i - 1] + self.h / 2) - (self.yRunge[i - 1] + self.h / 2 * k1)
        k3 = np.cos(self.x[i - 1] + self.h / 2) - (self.yRunge[i - 1] + self.h / 2 * k2)
        k4 = np.cos(self.x[i - 1] + self.h / 2) - (self.yRunge[i - 1] + self.h / 2 * k3)
        self.yRunge[i] = self.h / 6 * (k1 + k2 + k3 + k4) + self.yRunge[i - 1]
```

Exact solution:

```
def buildPlot(self):
    c = self.yStart*np.exp(self.xStart) - np.sin(self.xStart)*np.exp(self.xStart)/2 - np.cos(self.xStart)*np.exp(self.xStart)/2
    for i in range(0, self.count):
        self.y[i] = c * np.exp(-self.x[i]) + 0.5 * np.cos(self.x[i]) + 0.5 * np.sin(self.x[i])
```

Errors:

```
def errors(self):
    for i in range(0, self.count):
        self.errorEuler[i] = self.y[i] - self.yEuler[i]
        self.errorImprEuler[i] = self.y[i] - self.yImprEuler[i]
        self.errorRunge[i] = self.y[i] - self.yRunge[i]
```

Main program:

Firstly, enter the necessary values and create figure. Secondly, check the state of the slider. If there is a change, there is a need to run the update() function.

```
fig = plt.figure(figsize=(8, 7))
fig.subplots_adjust(left = 0.125, right = 0.9, bottom = 0.1, top = 0.9, wspace = 0.3, hspace = 0.6)
rcParams['axes.titlepad'] = 12

ax1 = fig.add_subplot(311)
ax2 = fig.add_subplot(312)
ax3 = fig.add_subplot(313)

axes_slider = plt.axes([0.29, 0.01, 0.45, 0.02])
slider = Slider(axes_slider, label='number of points', valmin=2, valmax=1000, valinit=100)

slider.on_changed(onChangeValue)
```

In this method, we create three subplots for numerical methods, exact solution, errors and new graphs that are built according to the new step.

```
graph = nm.NumMethods(xStart, xFinite, yStart, int(slider.val))
graph.buildPlot()
graph.eulerMethod()
graph.improvedEulerMethod()
graph.rungeKutta()
graph.errors()

...
ax1.set_title("\nNumerical Methods")
ax1.plot(graph.x, graph.yEuler, "blue", label="Euler Method")
ax1.legend(loc='upper right')
ax1.plot(graph.x, graph.yImprEuler, "cyan", label="Improved Euler")
ax1.legend(loc='upper right')
ax1.plot(graph.x, graph.yRunge, "red", label="Runge-Kutta")
ax1.legend(loc='lower left')

ax2.set_title("Exact solution")
ax2.plot(graph.x, graph.y, "black", label="Grapth")
ax2.legend(loc='upper center')
ax3.set_title("Errors")

ax3.plot(graph.x, graph.errorEuler, "blue", label="Euler Method")
ax3.plot(graph.x, graph.errorImprEuler, "cyan", label="Improved Euler")
ax3.plot(graph.x, graph.errorRunge, "red", label="Runge-Kutta")
plt.draw()
```