

# DA3450 - Numerical Methods for Business

Semester 6, 2023

## Individual Take-Home Coding Activity

Student ID: 206001F - aabidh.mm

## Instructions:

- Time allowed: **One day**
- Due on: **Sunday, December 17, 2023 at 4.00 am**
- Answer all **TWO (2)** questions.
- This is an open-book examination and accounts for 15% of the course assessment.
- Use this Colab Notebook to answer all questions. Once you finish, convert the Colab Notebook to a PDF file and upload it, along with your .ipynb file to the submission portal created on Moodle. **Email submissions are not allowed.**
- Use the webpage <https://onlineconvertfree.com/convert-format/ipynb-to-pdf/> to convert your .ipynb file to a PDF file.
- Type your Student ID number at the top of your .ipynb file.
- You must write your own codes and answers. Those should be well-documented and easy to read. If your codes and answers are similar (either partially or fully) to someone else's, this will be considered evidence of academic dishonesty, and you will be awarded a zero for the assignment.**
- Your python codes must be annotated. No points will be given if the code does not tell what it is doing.
- Be neat and legible.
- In case of any technical issues (eg: power cuts, laptop or Colab issues), you will not be given any excuse. Be smart to figure out ways to fix the problems.**

### Question 1:

Compute the sine integral

$$\int_0^{2\pi} \frac{\sin x}{x} dx$$

by using a very small value (for example,  $1 \times 10^{-12}$ ) instead of 0 for the lower limit and applying the following numerical methods:

- The trapezoid rule with 100 equally spaced sub-intervals over the whole interval  $[0, 2\pi]$ . Round the answer to 5 decimal digits.
- Simpson's rule with 100 equally spaced sub-intervals over the whole interval  $[0, 2\pi]$ . Round the answer to 5 decimal digits.

Compare each value obtained above to the value evaluated using `integrate.quad()` function. (Hint: Calculate relative approximate errors.)

```
In [1]: import numpy as np
from scipy import integrate

In [2]: # defining the function
def f(x):
    return np.sin(x) / x if x != 0 else 1

In [3]: #100 equally spaced sub-interval
n = 100

# over the whole interval [0,2π]
a = 1e-12 # very small value (for example, 1×10-12) instead of 0
b = 2 * np.pi

# Width interval
h = (b - a) / n

In [4]: # Trapezoid rule
Tintegral = 0.5 * (f(a) + f(b)) + sum(f(a + i * h) for i in range(1, n))

In [5]: # Multiplying by h
Tintegral *= h

In [6]: # Rounding answer to 5 decimal point
Tintegral = round(Tintegral, 5)

In [7]: print("trapezoid answer", Tintegral)

trapezoid answer 1.4182

In [8]: # Simpson's rule
Sintegral = f(a) + f(b) # Initialize the integral with a and b values

for i in range(1, n, 2):
    Sintegral += 4 * f(a + i * h) # case of odd indices, multiply by 4.

for i in range(2, n-1, 2):
    Sintegral += 2 * f(a + i * h) # case of even indices, Multiply by 2.

In [9]: # Multiply by h/3
Sintegral *= h / 3

# Rounding answer
Sintegral = round(Sintegral, 5)

print("Simpson's rule answer", Sintegral)

Simpson's rule answer 1.41815

In [10]: # Compute the integral using integrate
true_value, _ = integrate.quad(f, 1e-12, 2 * np.pi)
print(true_value)

1.4181515761316281

In [11]: # Calculate absolute errors
trapezoid_absolute_error = abs(Tintegral - true_value)
simpson_absolute_error = abs(Sintegral - true_value)

In [12]: trapezoid_absolute_error_rounded = round(trapezoid_absolute_error, 5)
simpson_absolute_error_rounded = round(simpson_absolute_error, 5)

print("Absolute error Trapezoid", trapezoid_absolute_error_rounded)
print("Absolute error Simpson's", simpson_absolute_error_rounded)

Absolute error Trapezoid 5e-05
Absolute error Simpson's 0.0

In [13]: # Calculate relative errors as %
def relative_error(approx, true):
    return abs(approx - true) / abs(true) * 100 # relative error

trapezoid_error = relative_error(Tintegral, true_value)
simpson_error = relative_error(Sintegral, true_value)

print("Relative error Trapezoid Rule", trapezoid_error,"%")
print("Relative error Simpson's Rule", simpson_error,"%")

Relative error Trapezoid Rule 0.003414576353246494 %
Relative error Simpson's Rule 0.0001113985660120649 %

In [14]: print("Relative error Trapezoid Rule {:.5f}%".format(trapezoid_error))
print("Relative error Simpson's Rule {:.5f}%".format(simpson_error))

Relative error Trapezoid Rule 0.00341%
Relative error Simpson's Rule 0.00011%
```

Both methods are providing highly accurate estimates of the integral, with Simpson's Rule slightly more accurate in this scenario.

### Question 2:

Consider solving the following IVP

$$\frac{dy}{dx} = yx^3 - 1.5y$$

over the interval from  $x = 0$  to  $2$  using a step size of  $0.25$ , where  $y(0) = 1$ .

The exact solution to this IVP is

$$y = e^{x^4/4 - 1.5x}$$

Use the following methods to numerically solve the given IVP and display all your results on the same graph including the exact solution. Compare those with its exact solution. Which method gives a better solution?

- Euler method
- Modified Euler method
- Second-order RK method

Euler method

```
In [15]: import numpy as np
import matplotlib.pyplot as plt

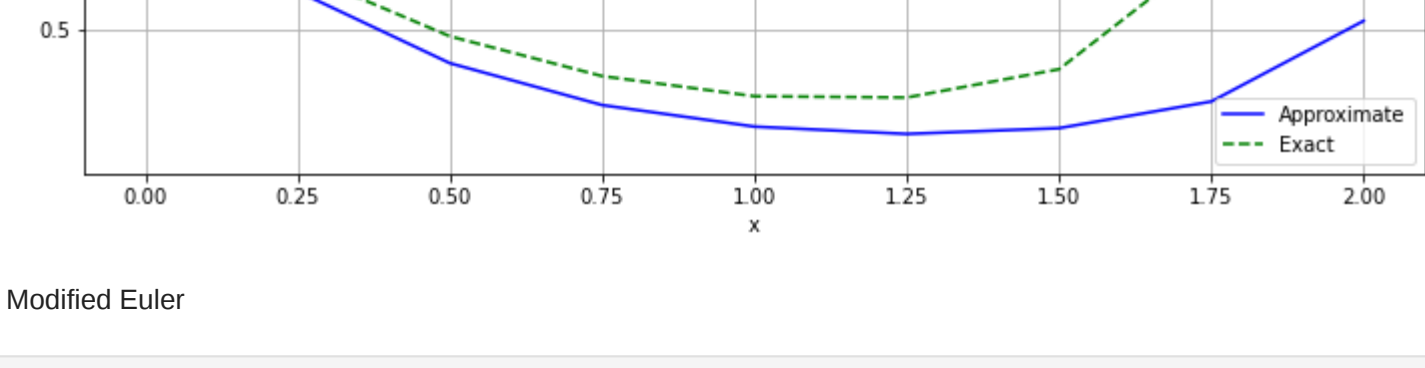
# Parameters
f = lambda x, y: y * x**3 - 1.5 * y # ODE
h = 0.25 # Step size
x = np.arange(0, 2.1, h) # Numerical grid
print(x)
y0 = 1 # Initial condition

# Euler method
y = np.zeros(len(x))
y[0] = y0

for i in range(0, len(y) - 1):
    y[i+1] = y[i] + h*f(x[i],y[i])
    print("i =", i, ", y[i+1] =", y[i+1])

plt.figure(figsize = (12, 8))
plt.plot(x, y, "b", label = "Approximate")
plt.plot(x, np.exp(x**4 / 4 - 1.5 * x), "g--", label = "Exact")
plt.title("Approximate and Exact Solution")
plt.xlabel("x")
plt.ylabel("y")
plt.grid()
plt.legend(loc = "lower right")
plt.show()

[0. 0.25 0.5 0.75 1. 1.25 1.5 1.75 2. ]
i = 0 y[i+1] = 0.625
i = 1 y[i+1] = 0.39306640625
i = 2 y[i+1] = 0.2579498291015625
i = 3 y[i+1] = 0.18842428922653198
i = 4 y[i+1] = 0.16487125307321548
i = 5 y[i+1] = 0.18354807471041568
i = 6 y[i+1] = 0.269586234738923
i = 7 y[i+1] = 0.529694828397087
```



Modified Euler

```
In [6]: # Parameters
f = lambda x, y: y * x**3 - 1.5 * y # ODE
h = 0.25 # Step size
x = np.arange(0, 2.1, h) # Numerical grid
y0 = 1 # Initial condition

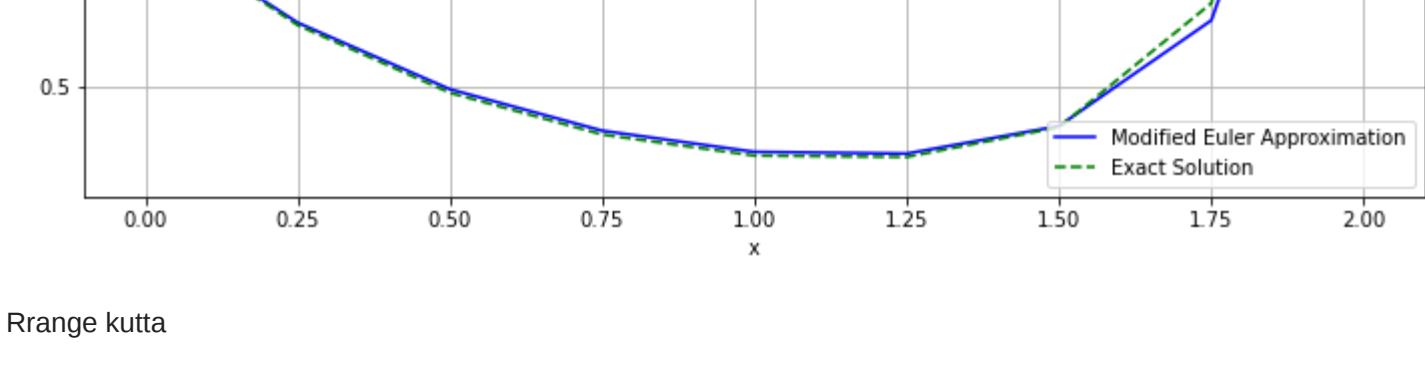
# Modified Euler method
y = np.zeros(len(x))
y[0] = y0

for i in range(0, len(y) - 1):
    xi_half = x[i] + 0.5 * h
    yi_half = y[i] + 0.5 * h * f(x[i], y[i])
    y[i + 1] = y[i] + h * f(xi_half, yi_half)

    print("i =", i, " y[i+1] =", y[i + 1])

# Plotting
plt.figure(figsize=(12, 8))
plt.plot(x, y, "b", label="Modified Euler")
plt.plot(x, np.exp(x**4 / 4 - 1.5 * x), "g--", label="Exact Solution")
plt.title("Modified Euler and Exact Solution")
plt.xlabel("x")
plt.ylabel("y")
plt.grid()
plt.legend(loc="lower right")
plt.show()

i = 0 y[i+1] = 0.69657099228515625
i = 1 y[i+1] = 0.49069589187274687
i = 2 y[i+1] = 0.363113917463176
i = 3 y[i+1] = 0.2979157079594314
i = 4 y[i+1] = 0.2925970834124262
i = 5 y[i+1] = 0.37758863143279575
i = 6 y[i+1] = 0.7028019355029231
i = 7 y[i+1] = 2.029022758906195
```



Runge kutta

```
In [19]: import numpy as np
import matplotlib.pyplot as plt

# Parameters
f = lambda x, y: y * x**3 - 1.5 * y # ODE
h = 0.25 # Step size
x = np.arange(0, 2.1, h) # Numerical grid
#print(x)
y0 = 1 # Initial condition

# RK2 method
y = np.zeros(len(x))
y[0] = y0

for i in range(0, len(y) - 1):
    k1 = f(x[i], y[i])
    k2 = f(x[i] + h, y[i] + h*k1)
    y[i+1] = y[i] + (h/2)*(k1+k2)
    print("i =", i, ", y[i+1] =", y[i+1])

i = 0 y[i+1] = 0.696533203125
i = 1 y[i+1] = 0.49200309183822614
i = 2 y[i+1] = 0.36392734046876285
i = 3 y[i+1] = 0.29826759813321213
i = 4 y[i+1] = 0.2944061785204611
i = 5 y[i+1] = 0.3879021820003594
i = 6 y[i+1] = 0.7536676122813379
i = 7 y[i+1] = 2.320435121699207
```

```
In [8]: # Parameters
f = lambda x, y: y * x**3 - 1.5 * y # ODE
h = 0.25 # Step size
x = np.arange(0, 2.1, h) # Numerical grid
y0 = 1 # Initial condition

# Euler method
y_euler = np.zeros(len(x))
y_euler[0] = y0

for i in range(0, len(y_euler) - 1):
    y_euler[i+1] = y_euler[i] + h*f(x[i], y_euler[i])

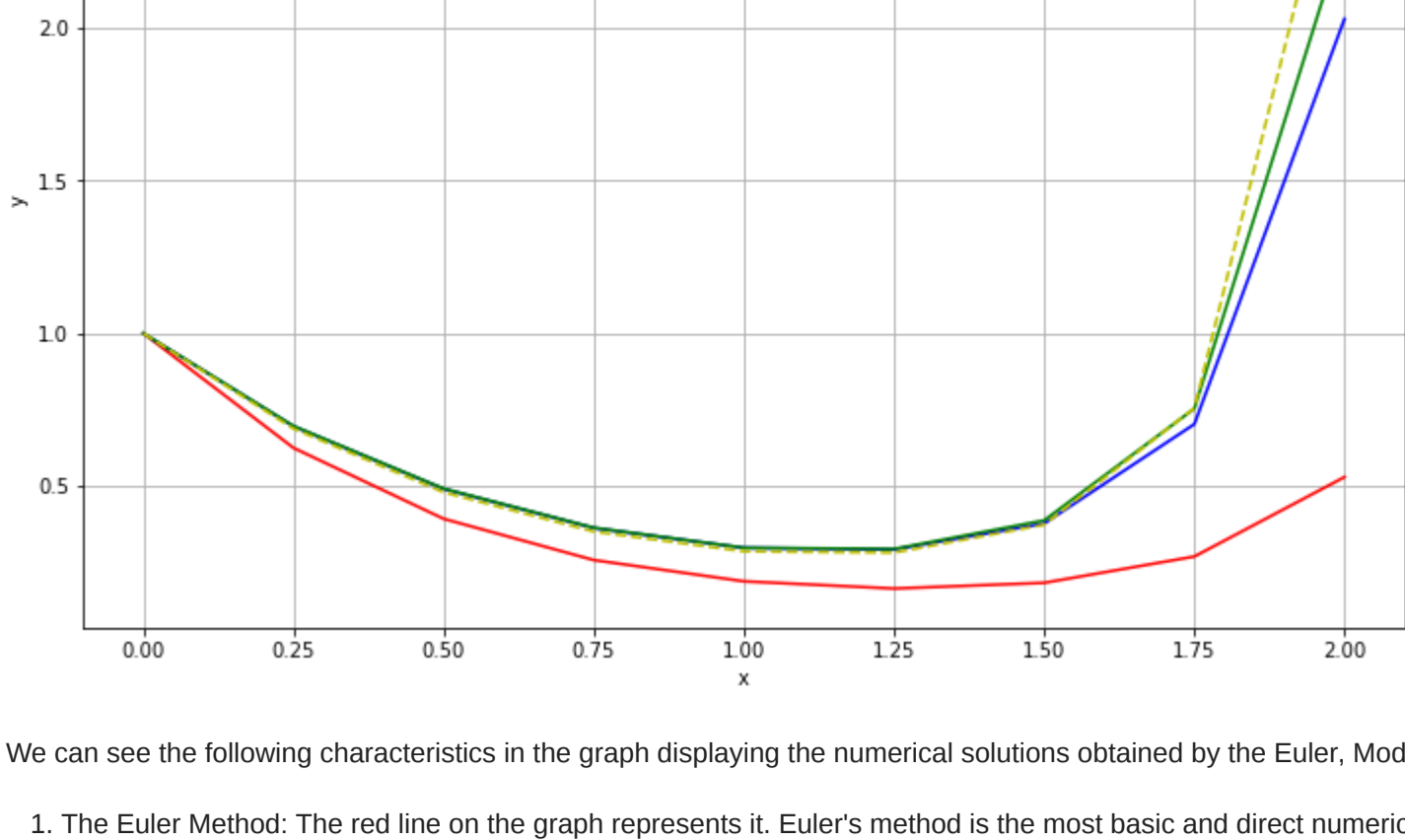
# Modified Euler method
y_modified_euler = np.zeros(len(x))
y_modified_euler[0] = y0

for i in range(0, len(y_modified_euler) - 1):
    xi_half = x[i] + 0.5 * h
    yi_half = y_modified_euler[i] + 0.5 * h * f(x[i], y_modified_euler[i])
    y_modified_euler[i + 1] = y_modified_euler[i] + h * f(xi_half, yi_half)

# RK2 method
y_rk2 = np.zeros(len(x))
y_rk2[0] = y0

for i in range(0, len(y_rk2) - 1):
    k1 = f(x[i], y_rk2[i])
    k2 = f(x[i] + h, y_rk2[i] + h * k1)
    y_rk2[i + 1] = y_rk2[i] + (h / 2) * (k1 + k2)

# Plotting
plt.figure(figsize=(12, 8))
plt.plot(x, y_euler, "r", label="Euler's Method")
plt.plot(x, y_modified_euler, "b", label="Modified Euler Method")
plt.plot(x, y_rk2, "g", label="RK2 Method")
plt.plot(x, np.exp(x**4 / 4 - 1.5 * x), "g--", label="Exact Solution")
plt.title("Approximation Methods vs Exact Solution")
plt.xlabel("x")
plt.ylabel("y")
plt.grid()
plt.legend()
plt.show()
```



We can see the following characteristics in the graph displaying the numerical solutions obtained by the Euler, Modified Euler, and RK2 methods :

- The Euler Method:** The red line on the graph represents it. Euler's method is the most basic and direct numerical method for solving differential equations. It does, however, accumulate significant error over iterations. This is visible in the graph, where the blue line (Euler's method) deviates significantly from the exact solution, particularly as x-values increase. Because of its inherent reliance on linear approximations over each step, the method deviates more from the exact solution.
- Modified Euler's Method:** The blue line on the graph represents this. The modified Euler's method improves on Euler's method by incorporating a midpoint evaluation to better approximate the function's behavior over each step.As shown in the graph, the green line (Modified Euler's method) has a much better alignment with the exact solution than Euler's
- RK2 Method :** The green line on the graph represents this. RK2, or the second-order Runge-Kutta method, improves accuracy by using a weighted average of two slopes within each step.

When compared to the other methods, the RK2 method consistently provides solutions that closely match the exact solution. Although the Modified Euler method produces a trajectory that is closer to the RK2 method than Euler's method, its accuracy falls short of the RK2 method.Finally, when these methods are compared, the hierarchy of accuracy places RK2 at the top, followed by the Modified Euler method, and finally, Euler's method. Choosing the RK2 method would be the best option for greater accuracy in approximating solutions for this particular differential equation.

```
In [ ]:
```