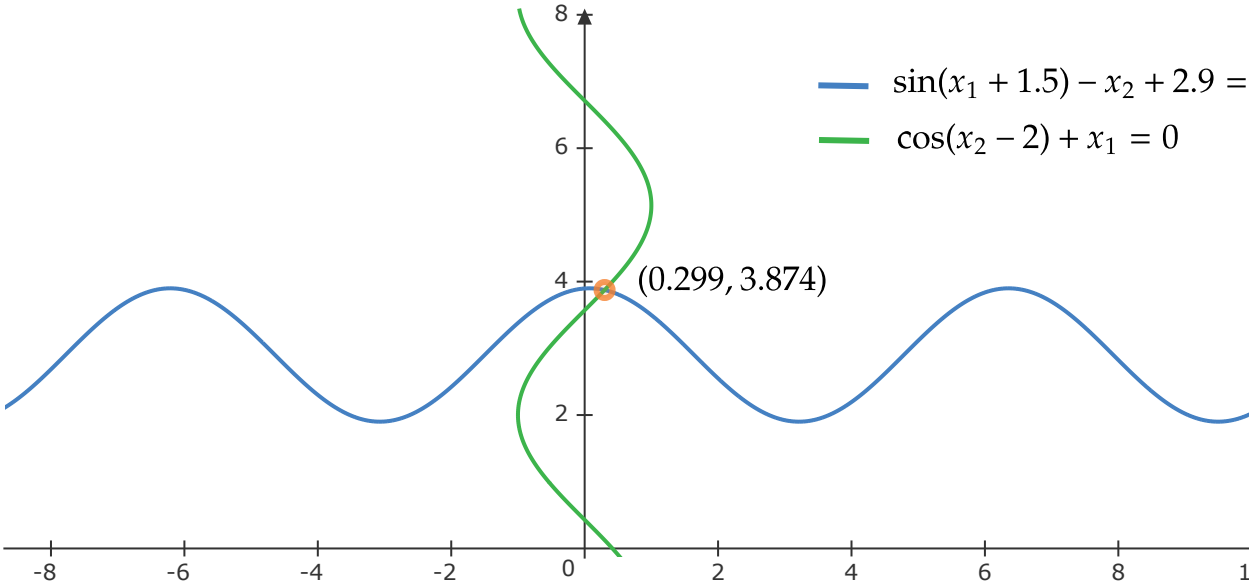
Assignment

Algorithmic and Computational Mathematics

For the system of nonlinear equations

$$\sin(x_1 + 1.5) - x_2 + 2.9 = 0$$
$$\cos(x_2 - 2) + x_1 = 0$$

1. Localise the roots of the system graphically.



The resulting graphs of the two curves of the two equations and indicate the points where they intersect, representing the possible roots of the system. The possible roots of the two curves should be $(0.299,\ 3.874)$.

2. Give the description of the Newton's method of solving systems of nonlinear equations.

Newton's method, also known as the Newton-Raphson method, is an iterative numerical technique used to solve systems of nonlinear equations. The Newton's method for solving systems of nonlinear equations is an extension of the one-dimensional Newton's method for finding roots of a single equation. Instead of solving a single equation, it aims to find the simultaneous solutions to a set of nonlinear equations. The method iteratively refines an initial guess to converge towards the true solution.

1. Formulate the system of equations:

Start with a system of equations in the form $\mathbf{F}(\mathbf{x}) = 0$, where $\mathbf{F}(\mathbf{x})$ represents a vector-valued function and $\mathbf{x}$ is the vector of unknowns.

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \ \mathbf{F}(\mathbf{x}) = \begin{pmatrix} f_1(x_1, x_2 \ \cdots \ x_n) \\ f_2(x_1, x_2 \ \cdots \ x_n) \\ \vdots \\ f_n(x_1, x_2 \ \ldots \ x_n) \end{pmatrix}$$

2. Choose an initial guess:

Start with an initial guess for the solution vector $\mathbf{x}^{(0)}$.

3. Evaluate the Jacobian matrix:

Calculate the Jacobian matrix $\mathbf{J}$, which contains the first-order partial derivatives of the vector function $\mathbf{F}(\mathbf{x})$ with respect to each variable in $\mathbf{x}$

$$\mathbf{J} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_1} & & \frac{\partial f_n}{\partial x_n} \end{pmatrix}$$

4. Solve the linear system:

Solve the linear system $\mathbf{J}\left(\mathbf{x}^{(k)}\right)\delta\mathbf{x}^{(k)} = -\mathbf{F}\left(\mathbf{x}^{(k)}\right)$, where $\mathbf{J}\left(\mathbf{x}^{(k)}\right)$ is the Jacobian matrix evaluated at the kth iteration, $\delta\mathbf{x}^{(k)}$ is the correction vector, and $\mathbf{F}\left(\mathbf{x}^{(k)}\right)$ is the vector function evaluated at the kth iteration.

Since $\mathbf{J}$ is a matrix, instead of dividing by $f'(x)$ we multiply by $\mathbf{J}^{-1}$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{J}^{-1}\left(\mathbf{x}^{(k)}\right)\mathbf{F}\left(\mathbf{x}^{(k)}\right)$$

$$\mathbf{J}\left(\mathbf{x}^{(k)}\right)\delta\mathbf{x}^{(k)} = -\mathbf{F}\left(\mathbf{x}^{(k)}\right)$$

where, $\delta\mathbf{x}^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$
using an initial guess $\mathbf{x}^{(0)}$.

5. Update the solution:

Update the solution estimate by computing $\mathbf{x}^{(k+1)} = \mathbf{x}^{k} + \delta\mathbf{x}^{k}$

6. Check convergence

Check if the solution has converged. If the change in the solution is below a specified tolerance or the maximum number of iterations is reached, terminate the process and consider $\mathbf{x}^{(k+1)}$ as the solution. Otherwise, go back to step 3 and repeat the process with the updated solution estimate.

7. Repeat steps 3 to 6:

Iterate until convergence is achieved or until the maximum number of iterations is reached.

3. Solve the system using Newton's method with the accuracy ε = 10−6.

$$f_1 = \sin(x_1 + 1.5) - x_2 + 2.9 = 0$$
$$f_2 = \cos(x_2 - 2) + x_1 = 0$$

In order to use Newton Raphson method, we must first determine the functional form of the partial derivatives.

$$J_{(1,1)} = \frac{\partial f_1}{\partial x_1} = \cos(x_1 + 1.5)$$
$$J_{(1,2)} = \frac{\partial f_1}{\partial x_2} = -1$$
$$J_{(2,1)} = \frac{\partial f_2}{\partial x_1} = 1$$
$$J_{(2,2)} = \frac{\partial f_2}{\partial x_2} = -\sin(x_2 - 2)$$

$$J = \begin{bmatrix} \cos(x_1 + 1.5) & -1 \\ 1 & -\sin(x_2 - 2) \end{bmatrix}$$

with $x_0 = [1, 2]^T$, we get

Formula

$$\begin{bmatrix} x_1^{(n+1)} \\ x_2^{(n+1)} \end{bmatrix} = \begin{bmatrix} x_1^n \\ x_2^n \end{bmatrix} - \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix}^{-1} \times f\left(\begin{bmatrix} x_1^n \\ x_2^n \end{bmatrix}\right)$$

$$n = 0, x_1 = 1, x_2 = 2$$

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} - \begin{bmatrix} -0.801144 & -1 \\ 1 & 0 \end{bmatrix}^{-1} \times f\left(\begin{bmatrix} 1 \\ 2 \end{bmatrix}\right)$$

$$= \begin{bmatrix} 1 \\ 2 \end{bmatrix} - \begin{bmatrix} -0.801144 & -1 \\ 1 & 0 \end{bmatrix}^{-1} \times \begin{bmatrix} 1.498472 \\ 2 \end{bmatrix}$$

$$= \begin{bmatrix} -1 \\ 5.100759 \end{bmatrix}$$

$$n = 1, x_1 = -1, y = 5.100759$$

$$\begin{bmatrix} -1 \\ 5.100759 \end{bmatrix} - \begin{bmatrix} 0.877583 & -1 \\ 1 & -0.040822 \end{bmatrix}^{-1} \times f\left(\begin{bmatrix} -1 \\ 5.100759 \end{bmatrix}\right)$$

$$= \begin{bmatrix} -1 \\ 5.100759 \end{bmatrix} - \begin{bmatrix} 0.877583 & -1 \\ 1 & -0.040822 \end{bmatrix}^{-1} \times \begin{bmatrix} -1.721334 \\ -1.999166 \end{bmatrix}$$

$$= \begin{bmatrix} 1.000568 \\ 5.135089 \end{bmatrix}$$

$$n = 2, x = 1.0000568,\ y = 5.135089$$

$$\begin{bmatrix} 1.000568 \\ 5.135089 \end{bmatrix} - \begin{bmatrix} -0.801483 & -1 \\ 1 & -0.801483 \end{bmatrix}^{-1} \times f\left(\begin{bmatrix} 1.000568 \\ 5.135089 \end{bmatrix}\right)$$

$$= \begin{bmatrix} 1.000568 \\ 5.135089 \end{bmatrix} - \begin{bmatrix} -0.801483 & -1 \\ 1 & -0.006504 \end{bmatrix}^{-1} \times \begin{bmatrix} -1.637072 \\ 0.000589 \end{bmatrix}$$

$$= \begin{bmatrix} 0.98939 \\ 3.506976 \end{bmatrix}$$

$$n = 3, x = 0.98939,\ y = 3.506976$$

$$\begin{bmatrix} 0.98939 \\ 3.506976 \end{bmatrix} - \begin{bmatrix} -0.794749 & -1 \\ 1 & -0.997964 \end{bmatrix}^{-1} \times f\left(\begin{bmatrix} 0.98939 \\ 3.506976 \end{bmatrix}\right)$$

$$= \begin{bmatrix} 0.98939 \\ 3.506976 \end{bmatrix} - \begin{bmatrix} -0.794749 & -1 \\ 1 & -0.997964 \end{bmatrix}^{-1} \times \begin{bmatrix} -0.000038 \\ 1.053167 \end{bmatrix}$$

$$= \begin{bmatrix} 0.402035 \\ 3.973738 \end{bmatrix}$$

$$n = 4,\ x = 0.402035, y = 3.973738$$

$$\begin{bmatrix} 0.402035 \\ 3.973738 \end{bmatrix} - \begin{bmatrix} -0.325215 & -1 \\ 1 & -0.919911 \end{bmatrix}^{-1} \times f\left(\begin{bmatrix} 0.402035 \\ 3.973738 \end{bmatrix}\right)$$

$$= \begin{bmatrix} 0.402035 \\ 3.973738 \end{bmatrix} - \begin{bmatrix} -0.325215 & -1 \\ 1 & -0.919911 \end{bmatrix}^{-1} \times \begin{bmatrix} -0.128098 \\ 0.009909 \end{bmatrix}$$

$$= \begin{bmatrix} 0.303705 \\ 3.877619 \end{bmatrix}$$

$$n = 5, x = 0.303705, y = 3.877619$$

$$\begin{bmatrix} 0.303705 \\ 3.877619 \end{bmatrix} - \begin{bmatrix} -0.230808 & -1 \\ 1 & -0.953298 \end{bmatrix}^{-1} \times f\left(\begin{bmatrix} 0.303705 \\ 3.877619 \end{bmatrix}\right)$$

$$= \begin{bmatrix} 0.303705 \\ 3.877619 \end{bmatrix} - \begin{bmatrix} -0.230808 & -1 \\ 1 & -0.953298 \end{bmatrix}^{-1} \times \begin{bmatrix} -0.004619 \\ 0.001674 \end{bmatrix}$$

$$= \begin{bmatrix} 0.298723 \\ 3.874149 \end{bmatrix}$$

$$n = 6, x = 0.298723, y = 3.874149$$

$$\begin{bmatrix} 0.298723 \\ 3.874149 \end{bmatrix} - \begin{bmatrix} -0.225959 & -1 \\ 1 & -0.95434 \end{bmatrix}^{-1} \times f\left(\begin{bmatrix} 0.298723 \\ 3.874149 \end{bmatrix}\right)$$

$$= \begin{bmatrix} 0.298723 \\ 3.874149 \end{bmatrix} - \begin{bmatrix} -0.225959 & -1 \\ 1 & -0.95434 \end{bmatrix}^{-1} \times \begin{bmatrix} -0.000012 \\ 0.000002 \end{bmatrix}$$

$$= \begin{bmatrix} 0.298712 \\ 3.874139 \end{bmatrix}$$

Approximate root using Newton Raphson method is
$x = 0.298712, y = 3.874139$

Python Code

```python
import math
import numpy as np

# Define the vector-valued function
def F(x):
    f1 = math.sin(x[0] + 1.5) - x[1] + 2.9
    f2 = math.cos(x[1] - 2) + x[0]
    return np.array([f1, f2])

# Define the Jacobian matrix
def Jacobian(x):
```

```
    j11 = math.cos(x[0] + 1.5)
    j12 = -1.0
    j21 = 1.0
    j22 = -math.sin(x[1] - 2)
    return np.array([[j11, j12], [j21, j22]])

# Solve the linear system J(x) * dx = -F(x) using numpy's linalg.solve
def SolveLinearSystem(J, F):
    return np.linalg.solve(J, -F)

# Perform Newton's method to solve the system of equations
def NewtonMethod(x, epsilon, maxIterations):
    for i in range(maxIterations):
        # Evaluate the function and Jacobian at the current point
        f = F(x)
        J = Jacobian(x)

        # Solve the linear system J(x) * dx = -F(x)
        dx = SolveLinearSystem(J, f)

        # Update the solution estimate
        x += dx

        # Check convergence
        norm = np.linalg.norm(dx)
        if norm < epsilon:
            print("Converged after", i+1, "iterations.")
            return

    print("Did not converge after", maxIterations, "iterations.")

# Set initial guess, epsilon, and maximum iterations
x = np.array([1.0, 2.0])
epsilon = 1e-6
maxIterations = 100

# Solve the system using Newton's method
NewtonMethod(x, epsilon, maxIterations)
```

```
# Print the solution with 6 decimal places
x1_formatted = "{:.6f}".format(x[0])
x2_formatted = "{:.6f}".format(x[1])
print("Solution: x1 =", x1_formatted, ", x2 =", x2_formatted)
```

Output:

```
$ python main.py
Converged after 8 iterations.
Solution: x1 = 0.298712 , x2 = 3.874139
```

C++ Code

```
#include <iostream>
#include <cmath>

using namespace std;

// Define the vector-valued function
void F(const double x1, const double x2, double& f1, double& f2) {
    f1 = sin(x1 + 1.5) - x2 + 2.9;
    f2 = cos(x2 - 2) + x1;
}

// Define the Jacobian matrix
void Jacobian(const double x1, const double x2, double& j11, double&
j12, double& j21, double& j22) {
    j11 = cos(x1 + 1.5);
    j12 = -1.0;
    j21 = 1.0;
    j22 = -sin(x2 - 2);
}

// Solve the linear system J(x) * Δx = -F(x) using Gaussian elimination
void SolveLinearSystem(const double j11, const double j12, const double
j21, const double j22,
```

```
                        const double f1, const double f2, double& dx1,
double& dx2) {
    double det = j11 * j22 - j12 * j21;
    dx1 = (j22 * f1 - j12 * f2) / det;
    dx2 = (-j21 * f1 + j11 * f2) / det;
}

// Perform Newton's method to solve the system of equations
void NewtonMethod(double& x1, double& x2, const double epsilon, const
int maxIterations) {
    double dx1, dx2;
    double f1, f2;
    double j11, j12, j21, j22;

    for (int i = 0; i < maxIterations; ++i) {
        // Evaluate the function and Jacobian at the current point
        F(x1, x2, f1, f2);
        Jacobian(x1, x2, j11, j12, j21, j22);

        // Solve the linear system J(x) * Δx = -F(x)
        SolveLinearSystem(j11, j12, j21, j22, f1, f2, dx1, dx2);

        // Update the solution estimate
        x1 -= dx1;
        x2 -= dx2;

        // Check convergence
        double norm = sqrt(dx1 * dx1 + dx2 * dx2);
        if (norm < epsilon) {
            cout << "Converged after " << i+1 << " iterations." << endl;
            return;
        }
    }

    cout << "Did not converge after " << maxIterations << "
iterations." << endl;
}
```

```
int main() {
    // Set initial guess, epsilon, and maximum iterations
    double x1 = 1.0;
    double x2 = 2.0;
    double epsilon = 1e-6;
    int maxIterations = 100;

    // Solve the system using Newton's method
    NewtonMethod(x1, x2, epsilon, maxIterations);

    // Print the solution
    cout << "Solution: x1 = " << x1 << ", x2 = " << x2 << endl;

    return 0;
}
```

Output

```
$ ./main.exe
Converged after 8 iterations.
Solution: x1 = 0.298712, x2 = 3.87414
```

4. Give the description of the gradient descent method of solving systems of nonlinear equations.

The gradient descent method is an iterative optimization algorithm commonly used to solve systems of nonlinear equations. It aims to find the root of a system by iteratively updating the initial guesses based on the gradients of the equations.

1. Define the system of equations:

Begin by defining a set of n nonlinear equations in n variables. The equations can be represented as $f(x) = 0$, where f is a vector-valued function $f = (f_1, f_2 \dots f_n)$ and $x = (x_1, x_2 \dots x_n)$ is the vector of variables.

2. Initialize the variables:

Choose initial guesses for the variables $(x_1, x_2 \dots x_n)$. These initial values will be

iteratively updated to converge to the root of the system.

3. Define the gradient vector:

Calculate the gradient vector (δf) for the system of equations. Each component of the gradient represents the partial derivative of the corresponding equation with respect to the variables.

4. Update the variables:

Use the gradient vector (δf) to update the variables iteratively. The update rule for each variable xi is given by:

$$x_{i(new)} = x_i - \text{learningRate} \times (\delta f)_i$$

Here, learningRate is a user-defined parameter called the learning rate or step size. It determines the size of each step taken during the optimization process. Choosing an appropriate learning rate is crucial for convergence.

5. Repeat steps 3 and 4:

Iterate steps 3 and 4 until a convergence criterion is met. Typically, the convergence criterion is defined based on the error between consecutive iterations. Common convergence criteria include checking if the norm of the gradient vector $(||\delta f||)$ is below a specified tolerance or if the change in variables $(||x_{new} - x||)$ is below a specified threshold.

6. Check convergence:

After the iterations, check if the algorithm has converged to a solution. If the maximum number of iterations is reached without satisfying the convergence criterion, the algorithm may not have converged to a root, or there may be multiple roots.

7. Output the root:

If the algorithm has converged, the final values of the variables represent the approximate root of the system of equations.

It's worth noting that the gradient descent method may not always converge or may converge to a local minimum instead of the global minimum, depending on the characteristics of the equations. Additional techniques like line search or other optimization algorithms may be necessary in some cases to improve convergence or find the global solution.

5. Solve the system using the gradient descent method with the accuracy 10^{-6}.

1. Define the system of equations:

$$f(x) = \sin(x + 1.5) - y + 2.9 = 0$$
$$g(y) = \cos(y - 2) + x = 0$$

2. Initialize the variables:
Let $x_0 = 1$ and $y_0 = 2$ be the initial guesses.

3. Define the gradient vectors:

The gradient vector for $f(x)$ is given by:

$$\delta f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} \cos(x + 1.5) \\ -1 \end{bmatrix}$$

The gradient vector for $g(y)$ is given by:

$$\delta g = \begin{bmatrix} \frac{\partial g}{\partial x} \\ \frac{\partial g}{\partial y} \end{bmatrix} = \begin{bmatrix} 1 \\ -\sin(y - 2) \end{bmatrix}$$

4. Update the variables using gradient descent:

Set a learning rate, α (step size), and iterate until convergence:
Repeat the following steps until $|f(x)| + |g(y)| < \varepsilon$:
Calculate the gradient at the current point:

$$\delta f_{eval} = \begin{bmatrix} \cos(x_{eval} + 1.5) \\ -1 \end{bmatrix}$$

$$\delta g_{eval} = \begin{bmatrix} 1 \\ -\sin(y_{eval} - 2) \end{bmatrix}$$

Update the variables:

$$x_{new} = x_{eval} - \alpha \times (\delta f_{eval(0)} + \delta g_{eval(0)})$$

$$y_{new} = y_{eval} - \alpha \times (\delta f_{eval(1)} + \delta_{eval(1)})$$

Update the current point:

$$x_{eval} = x_{new}$$
$$y_{eval} = y_{new}$$

5. Check convergence:

If $|f(x)| + |g(x)| < \varepsilon$, the algorithm has converged.

6. Output the solution:

The solution is $x = x_{eval}$ and $y = y_{eval}$

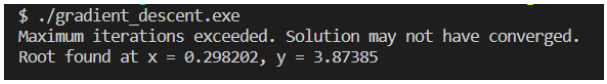
C++ Code

```cpp
#include <iostream>
#include <cmath>

double f1(double x, double y) {
    return sin(x + 1.5) - y + 2.9;
}

double f2(double x, double y) {
    return cos(y - 2) + x;
}

double f1_dx(double x, double y) {
    return cos(x + 1.5);
}

double f1_dy() {
    return -1.0;
}

double f2_dx() {
    return 1.0;
}
```

```
double f2_dy(double x, double y) {
    return -sin(y - 2);
}

const double learningRate = 0.01;
const int maxIterations = 1000;
const double tolerance = 1e-6;

void gradientDescent(double& x, double& y) {
    double error = 1.0;
    int iteration = 0;

    while (error > tolerance && iteration < maxIterations) {
        double x_new = x - learningRate * (f1(x, y) * f1_dx(x, y) +
f2(x, y) * f2_dx());
        double y_new = y - learningRate * (f1(x, y) * f1_dy() + f2(x,
y) * f2_dy(x, y));

        error = std::sqrt((x_new - x) * (x_new - x) + (y_new - y) *
(y_new - y));
        x = x_new;
        y = y_new;

        iteration++;
    }

    if (iteration >= maxIterations) {
        std::cout << "Maximum iterations exceeded. Solution may not
have converged." << std::endl;
    }
}

int main() {
    double x = 1.0;
    double y = 2.0;

    gradientDescent(x, y);
```

```
    std::cout << "Root found at x = " << x << ", y = " << y <<
std::endl;

    return 0;
}
```

Output



Python Code

```
import math

def f1(x, y):
    return math.sin(x + 1.5) - y + 2.9

def f2(x, y):
    return math.cos(y - 2) + x

def f1_dx(x, y):
    return math.cos(x + 1.5)

def f1_dy():
    return -1.0

def f2_dx():
    return 1.0

def f2_dy(x, y):
    return -math.sin(y - 2)

learning_rate = 0.01
max_iterations = 1000
tolerance = 1e-6

def gradient_descent():
    x = 1.0
```

```
    y = 2.0

    error = 1.0
    iteration = 0

    while error > tolerance and iteration < max_iterations:
        x_new = x - learning_rate * (f1(x, y) * f1_dx(x, y) + f2(x, y)
* f2_dx())
        y_new = y - learning_rate * (f1(x, y) * f1_dy() + f2(x, y) *
f2_dy(x, y))

        error = math.sqrt((x_new - x) * (x_new - x) + (y_new - y) *
(y_new - y))
        x = x_new
        y = y_new

        iteration += 1

    if iteration >= max_iterations:
        print("Maximum iterations exceeded. Solution may not have
converged.")

    return x, y

x_root, y_root = gradient_descent()
print(f"Root found at x = {x_root:.6f}, y = {y_root:.6f}")
```

Output

```
$ python gradient_descent.py
Maximum iterations exceeded. Solution may not have converged.
Root found at x = 0.298202, y = 3.873848
```

Question 2:

For the initial value Cauchy problem

$$xy' + y = 2y^2 \ln x, \quad y(1) = 1.5, \; x \in [1, 3]$$

1. Find the optimal integration step for the fourth-order Runge–Kutta method with the

accuracy 10^{-4}.

Solution:

The fourth-order Runge–Kutta method is given by:

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where, $k_1 = hf(x_n, y_n)$

$$k_2 = hf\left(x_n + \frac{1}{2}h,\ y_n + \frac{1}{2}k_1\right)$$

$$k_3 = hf\left(x_n + \frac{1}{2}h,\ y_n + \frac{1}{2}k_2\right)$$

$$k_4 = hf(x_n + h,\ y_n + k_3)$$

and h is the integration step size.

The local truncation error of this method is on the order of $O\left(h^5\right)$
which means that if we want the global error to be less than 10^{-4}
we need to choose, h such that $h^5 < 10^{-4}$

Solving for $h,$ we get

$$h < \left(10^{-4}\right)^{1/5} \approx 0.0398$$

Therefore, the optimal integration step for the fourth-order Runge-Kutta method with the accuracy, 10^{-4}

$h = 0.0398$ or smaller.

In this case, let's take optimal step size is 0.03

2. Find the solution using the fourth-order Runge–Kutta method with the accuracy 10^−4 and plot it.

The differential equation can be written as:

$$y' = \frac{2y^2 \ln x - y}{x}$$

The initial condition is:

$$y(1) = 1.5$$

The fourth-order Runge-Kutta method is:

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where,

$$k_1 = f(x_n, y_n)$$

$$k_2 = f\left(x_n + \frac{h}{2},\ y_n + \frac{h}{2}k_1\right)$$

$$k_3 = f\left(x_n + \frac{h}{2},\ y_n + \frac{h}{2}k_2\right)$$

$$k_4 = f(x_n + n, y_n + hk_3)$$

$$\text{and } f(x, y) = \frac{2y^2 \ln x - y}{x}$$

The step size is, $h = 0.03$

The accuracy is, $\epsilon = 10^{-4}$

To find the solution, we need to iterate the Runge=Kutta method until the error is less than the accuracy. The error can be estimated by:

$$e_n = \frac{|y_{n+1} - y_n|}{h}$$

The algorithm is:

- Initialize $x_0 = 1$ and $y_0 = 1.5$
- For $n = 0, 1, 2, \ldots$ until $x_n \geqslant 3$ or $e_n < \epsilon$
  - Calculate k_1, k_2, k_3, k_4 using the formulas above
  - Calculate y_{n+1} using the formula above
  - Calculate e_n using the formula above
  - If $e_n < \epsilon$, stop the iteration
  - Otherwise, set $x_{n+1} = x_n + h$ and $n = n + 1$

C++ Code

```cpp
#include <iostream>
#include <cmath>
#include <vector>

// Define the function f(x, y)
```

```
double f(double x, double y) {
    return (2 * pow(y, 2) * log(x) - y) / x;
}

int main() {
    // Define the initial values and parameters
    double x0 = 1;
    double y0 = 1.5;
    double h = 0.03;
    double eps = 1e-4;

    // Initialize the vectors to store the values of x and y
    std::vector<double> x_list;
    std::vector<double> y_list;

    // Initialize the error
    double error = INFINITY;

    // Append initial values to the lists
    x_list.push_back(x0);
    y_list.push_back(y0);

    // Iterate the Runge-Kutta method until the error is less than eps
or x >= 3
    while (error >= eps && x_list.back() < 3) {
        // Calculate k1, k2, k3, k4
        double k1 = f(x_list.back(), y_list.back());
        double k2 = f(x_list.back() + h / 2, y_list.back() + h / 2 *
k1);
        double k3 = f(x_list.back() + h / 2, y_list.back() + h / 2 *
k2);
        double k4 = f(x_list.back() + h, y_list.back() + h * k3);

        // Calculate y_next
        double y_next = y_list.back() + h / 6 * (k1 + 2 * k2 + 2 * k3 +
k4);

        // Calculate error
        error = std::abs(y_next - y_list.back()) / h;
```

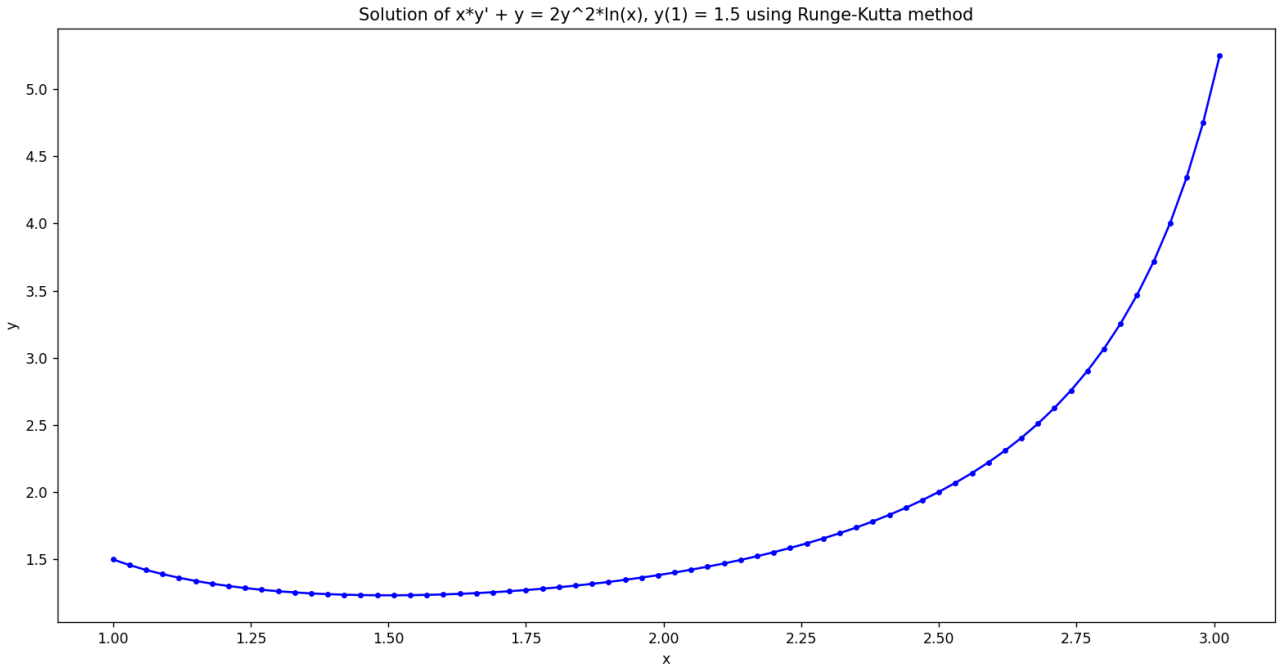
```
        // Append x_next and y_next to the lists
        double x_next = x_list.back() + h;
        x_list.push_back(x_next);
        y_list.push_back(y_next);
    }

    // Print x_list, y_list
    for (int i = 0; i < x_list.size(); ++i) {
        std::cout << x_list[i] << ", " << y_list[i] << std::endl;
    }

    return 0;
}
```

Python Code

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function f(x, y)
def f(x, y):
    return (2 * y**2 * np.log(x) - y) / x

# Define the initial values and parameters
x0 = 1
y0 = 1.5
h = 0.03
eps = 10**(-4)

# Initialize the lists to store the values of x and y
x_list = [x0]
y_list = [y0]

# Initialize the error
error = np.inf
```

```
# Iterate the Runge-Kutta method until the error is less than eps or x
>= 3
while error >= eps and x_list[-1] < 3:
    # Calculate k1, k2, k3, k4
    k1 = f(x_list[-1], y_list[-1])
    k2 = f(x_list[-1] + h / 2, y_list[-1] + h / 2 * k1)
    k3 = f(x_list[-1] + h / 2, y_list[-1] + h / 2 * k2)
    k4 = f(x_list[-1] + h, y_list[-1] + h * k3)

    # Calculate y_next
    y_next = y_list[-1] + h / 6 * (k1 + 2 * k2 + 2 * k3 + k4)

    # Calculate error
    error = abs(y_next - y_list[-1]) / h

    # Append x_next and y_next to the lists
    x_next = x_list[-1] + h
    x_list.append(x_next)
    y_list.append(y_next)

# Plot the solution
plt.plot(x_list, y_list, 'b.-')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Solution of x*y\' + y = 2y^2*ln(x), y(1) = 1.5 using Runge-
Kutta method')
plt.show()
```

Output



3. **Find the solution using the Euler method and the optimal step found above and add its plot to the graph.**

Solution:

The idea of the Euler method is to start from the initial point (x_0, y_0) and use the slope of the curve at that point. which is given by $f(x_0, y_0)$, to find the next point (x_1, y_1) on the curve. The next point is obtained by moving a small distance h along the x-axis and then moving up or down by h. $f(x_0, y_0)$ along the y-axis. This process is repeated until the desired value of x is reached.

The formula for finding the next point (x_{n+1}, y_{n+1}) from the previous point (x_n, y_n) is

$$x_{n+1} = x_n + h$$
$$y_{n+1} = y_n + h \cdot f(x_n, y_n)$$

where h is the step size, which determines how many points are used to approximate the curve. The smaller the step size, the more accurate the approximation.

The algorithm of the Euler method can be summarised as follows:

- Define the function $f(x, y)$ that represents the differential equation.
- Define the initial values x_0 and y_0 and the interval of interest $[x_0, x_f]$
- Create an array of x values from x_0 to x_f with step size h

- Create an array of y values with the same length as x, and assign the initial value to the first element
- Use a for loop to iterate over the x array, starting from the second element, and apply the formula to update the corresponding y value
- Plot or return the x and y arrays as the approximate solution

C++ Code

```cpp
#include <cmath>
#include "matplotlibcpp.h"
#include "xtensor/xarray.hpp"
#include "xtensor/xview.hpp"
#include "xtensor/xio.hpp"

namespace plt = matplotlibcpp;

// Define the function that represents the differential equation
double f(double x, double y) {
    return (2*std::pow(y,2)*std::log(x) - y)/x;
}

int main() {
    // Define the initial values and the interval of interest
    double x0 = 1;
    double y0 = 1.5;
    double xf = 3;
    double h = 0.03;

    // Create an array of x values from x0 to xf with step size h
    xt::xarray<double> x = xt::arange(x0, xf+h, h);

    // Create an array of y values with the same length as x
    // Assign the initial value to the first element
    xt::xarray<double> y = xt::zeros(x.shape());
    y(0) = y0;

    // Use a for loop to iterate over the x array
    // Apply the Euler formula to update the corresponding y value
    for (size_t i = 1; i < x.size(); i++) {
```

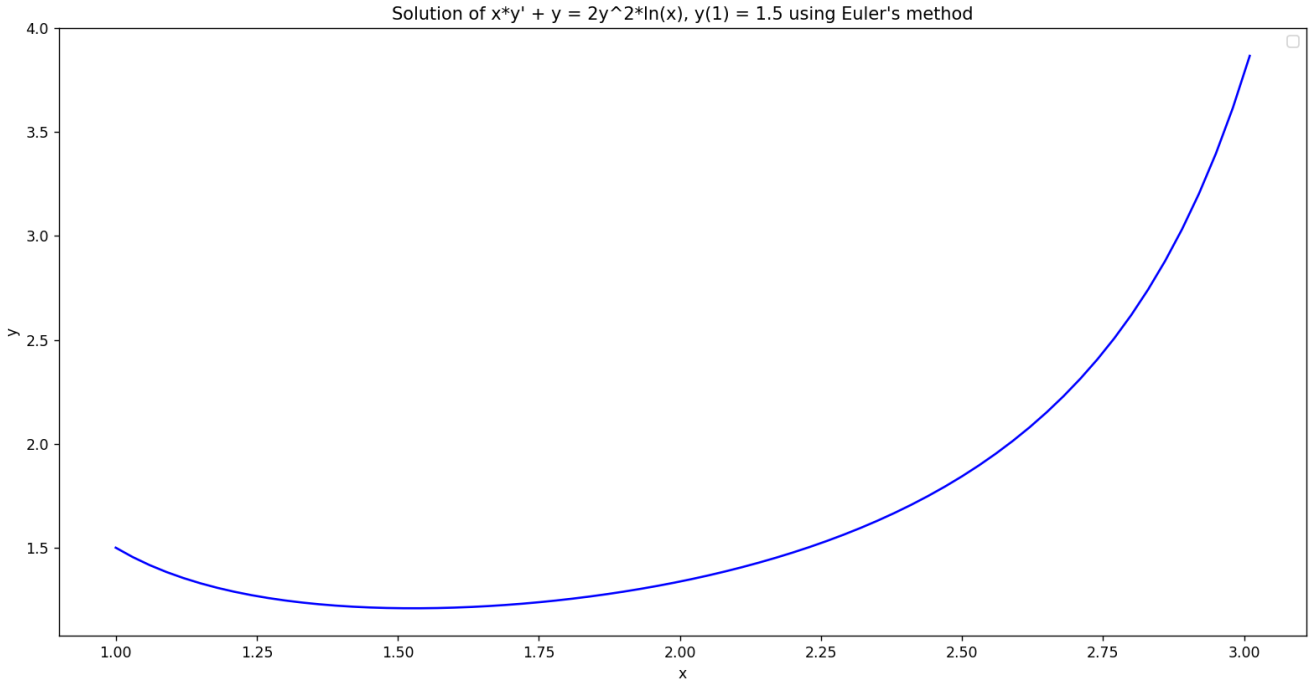
```
        y(i) = y(i-1) + h*f(x(i-1), y(i-1));
    }

    // Plot the x and y arrays using matplotlibcpp
    plt::plot(x, y, "b-");
    plt::xlabel("x");
    plt::ylabel("y");
    plt::title("Solution of x*y' + y = 2y^2*ln(x), y(1) = 1.5 using
Euler's method");
    plt::legend();
    plt::show();
}
```

Python Code

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function that represents the differential equation
f = lambda x, y: ( 2*np.power(y,2)*np.log(x) - y )/x

# Define the initial values and the interval of interest
x0 = 1
y0 = 1.5
xf = 3
h = 0.03

# Create an array of x values from x0 to xf with step size h
x = np.arange(x0, xf+h, h)

# Create an array of y values with the same length as x
# Assign the initial value to the first element
y = np.zeros(len(x))
y[0] = y0

# Use a for loop to iterate over the x array
# Apply the Euler formula to update the corresponding y value
```

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

# Plot the x and y arrays using matplotlib.pyplot
plt.plot(x, y, 'b-', label='Euler')
plt.xlabel('x')
plt.ylabel('y')
plt.title("Solution using Euler's method")
plt.legend()
plt.show()
```

Output



4. Find the exact solution of the problem, calculate the maximal absolute deviation from the approximate solutions at the integration points.

$$\text{Given, } xy' + y = 2y^2 \ln x$$

$$\text{can also be written as } \frac{1}{y^2} \cdot \frac{dy}{dx} + \frac{1}{xy} = \frac{\ln x}{x} \qquad (1)$$

Let, $\frac{1}{y} = t$

Differentiating w.r.t x

$$-\frac{1}{y^2} \cdot \frac{dy}{dx} = \frac{dt}{dx}$$

From (1) we get,

$$-\frac{dt}{dx} + \frac{1}{x} \cdot t = \frac{\ln x}{x}$$

$$or\ \frac{dt}{dx} - \frac{t}{x} = -\frac{\ln x}{x}$$

Integration Factor, $IF = e^{\int -\frac{dx}{x}} = e^{-\ln x} = x^{-1} = \frac{1}{x}$

$$\Longrightarrow t \times \frac{1}{x} = \int -\frac{\ln x}{x} \cdot \frac{1}{x} dx + C$$

Let $v = \ln x,\ \frac{1}{x} dx = dv$

$$= -\int v \cdot e^{-v} dv + C$$

$$= -\left(v \cdot \left(-e^{v}\right) + \int 1.e^{-v} dv\right) + C$$

$$\frac{t}{x} = v.e^{-v} + e^{v} + C$$

$$\frac{t}{x} = e^{-v}(v+1) + C$$

$$\frac{1}{xy} = \frac{1}{x}(1 + \ln x) + C$$

$$\frac{1}{y} = 1 + \ln x + Cx$$

Given, $y(1) = 1.5$

$$C = -\frac{1}{3}$$

$$y = \frac{1}{1 + \ln x - \frac{x}{3}}$$

C++ Code

```cpp
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

// Define the function f(x,y) = (2y^2*ln(x) - y)/x
double f(double x, double y) {
  return (2*y*y*log(x) - y)/x;
}

// Define the exact solution y(x) = 1/(1 + ln(x) - x/3)
double y_exact(double x) {
  return 1/(1 + log(x) - x/3);
}

// Define the Euler method
double euler(double x0, double y0, double h, double x) {
  double y = y0;
  while (x0 < x) {
    y = y + h*f(x0, y);
    x0 = x0 + h;
  }
  return y;
}
```

```
// Define the Runge-Kutta method of order 4
double rk4(double x0, double y0, double h, double x) {
  double k1, k2, k3, k4;
  double y = y0;
  while (x0 < x) {
    k1 = h*f(x0, y);
    k2 = h*f(x0 + h/2, y + k1/2);
    k3 = h*f(x0 + h/2, y + k2/2);
    k4 = h*f(x0 + h, y + k3);
    y = y + (k1 + 2*k2 + 2*k3 + k4)/6;
    x0 = x0 + h;
  }
  return y;
}

int main() {
  // Initial values
  double x0 = 1.0;
  double y0 = 1.5;

  // Step size
  double h = 0.03;

  // Upper bound
  double b = 3.0;

  // Initialize the maximal deviation variables
  double max_dev_euler = 0.0;
  double max_dev_rk4 = 0.0;

  // Loop over the values of x
  for (double x = x0; x <= b; x += h) {
    double ye = euler(x0, y0, h, x); // Euler solution
    double yr = rk4(x0, y0, h, x); // RK4 solution
    double yt = y_exact(x); // Exact solution
    double ee = abs(ye - yt); // Error of Euler
    double er = abs(yr - yt); // Error of RK4
```

```
    // Update the maximal deviation variables
    if (ee > max_dev_euler) {
      max_dev_euler = ee;
    }
    if (er > max_dev_rk4) {
      max_dev_rk4 = er;
    }
  }

  // Print the maximal deviation values
  cout << "Maximal deviation of Euler method: " << max_dev_euler <<
"\n";
  cout << "Maximal deviation of RK4 method: " << max_dev_rk4 << "\n";

  return 0;
}
```

Output

```
$ ./a.exe
Maximal deviation of Euler method: 2.70593
Maximal deviation of RK4 method: 3.84003
```

5. **Put the results of calculations in a table and on the graph, compare them.**

Code to Compare The Error (C++)

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

// Define the function f(x,y) = (2y^2*ln(x) - y)/x
double f(double x, double y) {
  return (2*y*y*log(x) - y)/x;
}

// Define the exact solution y(x) = 1/(1 + ln(x) - x/3)
double y_exact(double x) {
```

```
  return 1/(1 + log(x) - x/3);
}

// Define the Euler method
double euler(double x0, double y0, double h, double x) {
  double y = y0;
  while (x0 < x) {
    y = y + h*f(x0, y);
    x0 = x0 + h;
  }
  return y;
}

// Define the Runge-Kutta method of order 4
double rk4(double x0, double y0, double h, double x) {
  double k1, k2, k3, k4;
  double y = y0;
  while (x0 < x) {
    k1 = h*f(x0, y);
    k2 = h*f(x0 + h/2, y + k1/2);
    k3 = h*f(x0 + h/2, y + k2/2);
    k4 = h*f(x0 + h, y + k3);
    y = y + (k1 + 2*k2 + 2*k3 + k4)/6;
    x0 = x0 + h;
  }
  return y;
}

// Define the function to print the table
void print_table(double x0, double y0, double h, double b) {
  cout << "x\t\tExact\t\tEuler\t\tRK4\t\tError(Euler)\tError(RK4)\n";
  cout << fixed << setprecision(6);
  for (double x = x0; x <= b; x += h) {
    double ye = euler(x0, y0, h, x); // Euler solution
    double yr = rk4(x0, y0, h, x); // RK4 solution
    double yt = y_exact(x); // Exact solution
    double ee = abs(ye - yt); // Error of Euler
    double er = abs(yr - yt); // Error of RK4
```

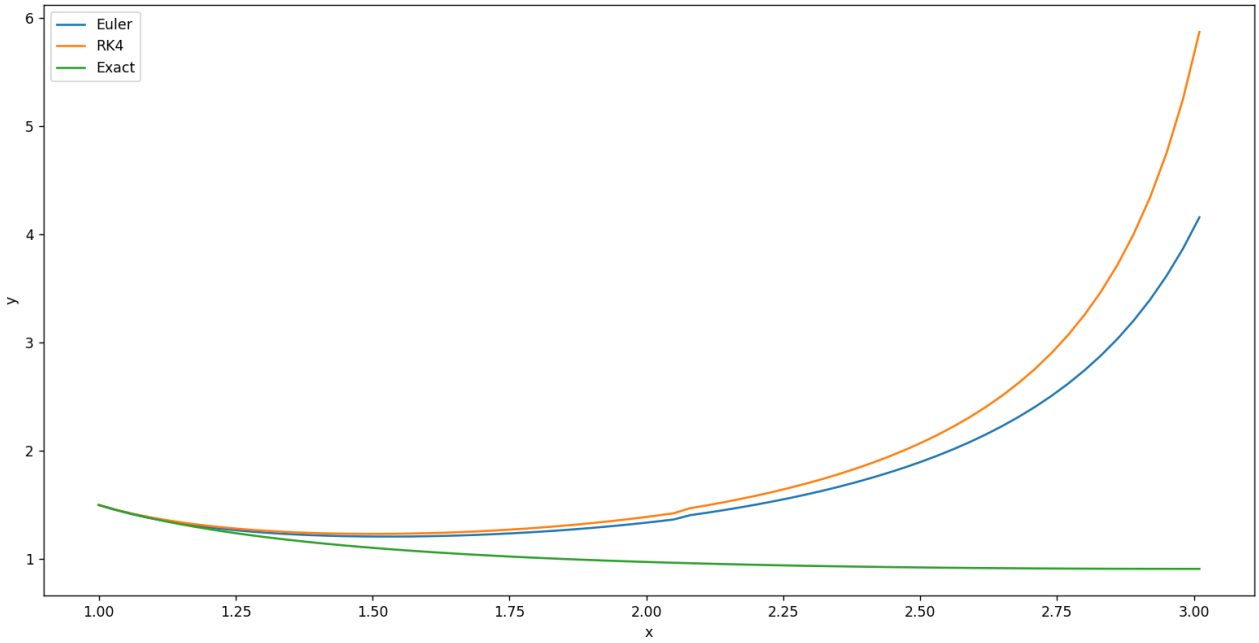
```
    cout << x << "\t" << yt << "\t" << ye << "\t" << yr << "\t" << ee
<< "\t\t" << er << "\n";
  }
}

int main() {
  // Initial values
  double x0 = 1.0;
  double y0 = 1.5;

  // Step size
  double h = 0.03;

  // Upper bound
  double b = 3.0;

  // Print the table
  print_table(x0, y0, h, b);

  return 0;
}
```

Output

```
$ ./main.exe
x               Exact           Euler           RK4             Error(Euler)    Error(RK4)
1.000000        1.500000        1.500000        1.500000        0.000000                0.000000
1.030000        1.457247        1.455000        1.458185        0.002247                0.000938
1.060000        1.418569        1.416267        1.422061        0.002303                0.003492
1.090000        1.383424        1.382799        1.390778        0.000624                0.007354
1.120000        1.351360        1.353811        1.363654        0.002451                0.012294
1.150000        1.322002        1.328676        1.340140        0.006674                0.018138
1.180000        1.295033        1.306888        1.319791        0.011855                0.024758
1.210000        1.270185        1.288036        1.302240        0.017851                0.032056
1.240000        1.247228        1.271783        1.287184        0.024555                0.039956
1.270000        1.225966        1.257849        1.274372        0.031883                0.048406
1.300000        1.206228        1.246002        1.263591        0.039775                0.057364
1.330000        1.187866        1.236048        1.254666        0.048182                0.066800
1.360000        1.170753        1.227823        1.247449        0.057070                0.076696
1.390000        1.154774        1.221189        1.241813        0.066415                0.087039
1.420000        1.139830        1.216031        1.237654        0.076200                0.097823
1.450000        1.125834        1.212250        1.234882        0.086415                0.109048
1.480000        1.112708        1.209763        1.233425        0.097055                0.120717
1.510000        1.100381        1.208501        1.233219        0.108121                0.132838
1.540000        1.088792        1.208407        1.234215        0.119615                0.145423
1.570000        1.077886        1.209432        1.236372        0.131546                0.158487
```

1.600000	1.067612	1.211537	1.239658	0.143926	0.172047
1.630000	1.057925	1.214691	1.244049	0.156766	0.186124
1.660000	1.048785	1.218871	1.249528	0.170086	0.200743
1.690000	1.040155	1.224058	1.256086	0.183903	0.215931
1.720000	1.032001	1.230242	1.263719	0.198241	0.231717
1.750000	1.024294	1.237417	1.272429	0.213123	0.248136
1.780000	1.017004	1.245583	1.282227	0.228579	0.265222
1.810000	1.010108	1.254745	1.293125	0.244638	0.283017
1.840000	1.003581	1.264914	1.305146	0.261333	0.301565
1.870000	0.997402	1.276104	1.318316	0.278703	0.320914
1.900000	0.991551	1.288337	1.332668	0.296785	0.341117
1.930000	0.986012	1.301638	1.348242	0.315626	0.362230
1.960000	0.980766	1.316037	1.365084	0.335271	0.384318
1.990000	0.975799	1.331573	1.383248	0.355774	0.407449
2.020000	0.971096	1.348286	1.402797	0.377190	0.431701
2.050000	0.966645	1.366227	1.423799	0.399583	0.457155
2.080000	0.962432	1.385450	1.446336	0.423018	0.483904
2.110000	0.958447	1.406018	1.470496	0.447572	0.512049
2.140000	0.954679	1.428003	1.496382	0.473324	0.541703
2.170000	0.951118	1.451482	1.524109	0.500363	0.572991
2.200000	0.947756	1.476545	1.553805	0.528789	0.606049
2.230000	0.944583	1.503292	1.585616	0.558709	0.641033
2.260000	0.941592	1.531833	1.619705	0.590241	0.678113
2.290000	0.938775	1.562294	1.656259	0.623519	0.717484
2.320000	0.936125	1.594813	1.695487	0.658688	0.759362
2.350000	0.933635	1.629547	1.737627	0.695912	0.803992
2.380000	0.931301	1.666672	1.782952	0.735372	0.851651
2.410000	0.929115	1.706385	1.831770	0.777271	0.902655
2.440000	0.927072	1.748910	1.884436	0.821838	0.957364
2.470000	0.925168	1.794497	1.941359	0.869329	1.016191
2.500000	0.923397	1.843433	2.003012	0.920036	1.079615
2.530000	0.921756	1.896043	2.069942	0.974287	1.148186
2.560000	0.920239	1.952697	2.142790	1.032457	1.222551
2.590000	0.918844	2.013820	2.222309	1.094976	1.303465
2.620000	0.917565	2.079901	2.309387	1.162336	1.391822
2.650000	0.916400	2.151506	2.405083	1.235105	1.488683
2.680000	0.915346	2.229290	2.510668	1.313944	1.595323
2.710000	0.914398	2.314019	2.627679	1.399621	1.713281
2.740000	0.913555	2.406595	2.757993	1.493040	1.844437
2.770000	0.912814	2.508080	2.903926	1.595267	1.991113
2.800000	0.912171	2.619741	3.068373	1.707570	2.156203
2.830000	0.911624	2.743093	3.254995	1.831469	2.343370
2.860000	0.911171	2.879971	3.468488	1.968800	2.557317
2.890000	0.910811	3.032610	3.714989	2.121799	2.804178
2.920000	0.910539	3.203761	4.002657	2.293222	3.092117
2.950000	0.910356	3.396849	4.342592	2.486493	3.432236
2.980000	0.910258	3.616186	4.750288	2.705928	3.840030

Comparison In One Plot



Python Code (Plot)

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the function f(x,y) = (2y^2*ln(x) - y)/x
def f(x, y):
  return (2*y*y*np.log(x) - y)/x

# Define the exact solution y(x) = 1/(1 + ln(x) - x/3)
def y_exact(x):
  return 1/(1 + np.log(x) - x/3)

# Define the Euler method
def euler(x0, y0, h, x):
  y = y0
  while x0 < x:
    y = y + h*f(x0, y)
    x0 = x0 + h
  return y

# Define the Runge-Kutta method of order 4
def rk4(x0, y0, h, x):
```

```
  y = y0
  while x0 < x:
    k1 = h*f(x0, y)
    k2 = h*f(x0 + h/2, y + k1/2)
    k3 = h*f(x0 + h/2, y + k2/2)
    k4 = h*f(x0 + h, y + k3)
    y = y + (k1 + 2*k2 + 2*k3 + k4)/6
    x0 = x0 + h
  return y

# Initial values
x0 = 1.0
y0 = 1.5

# Step size
h = 0.03

# Upper bound
b = 3.0

# Create an array of x values
x = np.arange(x0, b+h, h)

# Create arrays of y values using the methods
y_euler = [euler(x0, y0, h, xi) for xi in x]
y_rk4 = [rk4(x0, y0, h, xi) for xi in x]
y_true = [y_exact(xi) for xi in x]

# Plot the graphs
plt.plot(x, y_euler, label='Euler')
plt.plot(x, y_rk4, label='RK4')
plt.plot(x, y_true, label='Exact')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```