数値計算-後期中間レポート

HI4 45 番 山口惺司

2024 年 11 月 18 日

課題 3.1

3.1.a 問題:

次の 1 解微分方程式をオイラー法で解け. ただし, $h = 0.1, 0 \leq x \leq 2.5$ とせよ.

$$\frac{dy}{dx} = xy, \text{初期条件 } x = 0 \text{ で } y = 1 \tag{1}$$

得られた結果を解析解 $y = e^{\frac{x^2}{2}}$ とともにグラフで表し比較せよ.

3.1.b アルゴリズム:

オイラー法のアルゴリズムを図 1 に示す.

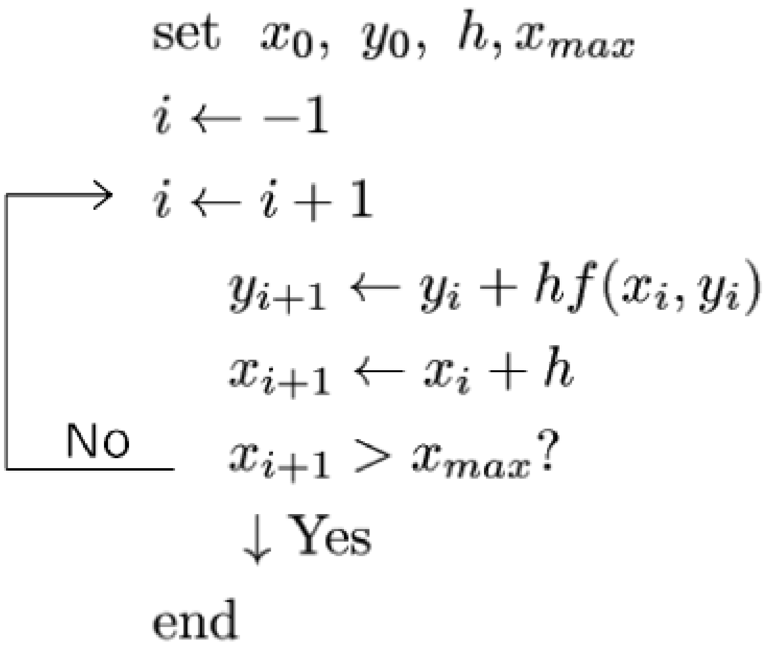

図 1　3.1 オイラー法アルゴリズム

3.1.c プログラムリスト:

使用したプログラムを以下に示す.

ソースコード 1　演習課題 3.1 Python プログラム

```
#3-1
import numpy as np
import math
import sympy as sym
import matplotlib.pyplot as plt
import japanize_matplotlib

def EulerMethod(x_val, y_val, h, x_max, f, x, y):
    i = 0
    while(True):
        y_val.append(y_val[i] + h * f.subs([(x, x_val[i]), (y, y_val[i])]))
        x_val.append(x_val[i] + h)
        if(x_val[i+1] > x_max):
            break;
        i += 1

def error(est, act): #誤差率を求める関数
    return abs(100 * (est - act) / act)

def main():
    x_val = [0]
    y_val = [1]
    h = 0.1
    x_max = 2.5
    x = sym.symbols("x")
    y = sym.symbols("y")
    f = x * y
    EulerMethod(x_val, y_val, h, x_max, f, x, y)

    expr = sym.exp(x**2 / 2)
    expr_func = sym.lambdify(x, expr, "numpy")
    px = np.linspace(0, x_max, 100)
    py = expr_func(px)
    plt.plot(px, py, color = "blue", label = '与えられた解析解')

    #近似曲線
    plt.plot(x_val, y_val, 'r--', label = '得られた結果')
    plt.xlim(0, x_max)
    plt.ylim(0, 25)
    plt.legend(loc = "upper left")

    plt.xlabel("x")
```

```python
    plt.ylabel("y")

    plt.show()

    #誤差率
    px = np.linspace(0, x_max, int(x_max/h+1))
    py = expr_func(px)
    print(" x | 誤差率 ")
    print("--------------")
    for i in range(len(py)):
        print(f"{px[i]:.1f}", f"{round(error(py[i], y_val[i]), 3):6.3f}", end = "%\n", sep = " | ")


if __name__ == "__main__":
    main()
```

3.1.d 実行結果:

実行結果を図 2 に示す.

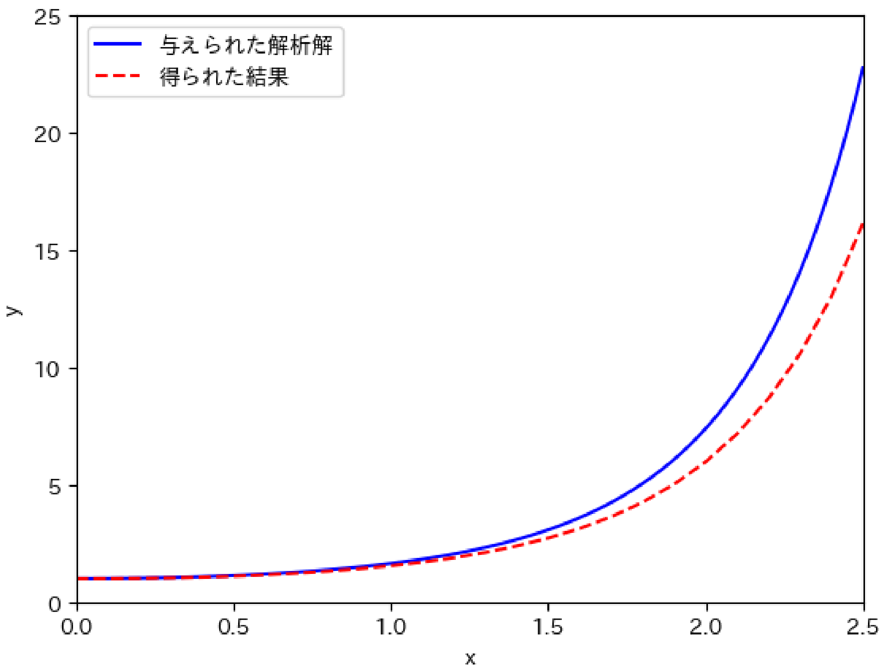

図 2　3.1 実行結果

3.1.e 考察:

図 2 から得られた解析解と与えられた解析解は似たような曲線になっていることが分かる.
また出力した局所誤差を以下に示す.

局所誤差

x	誤差率
0.0	0.000%
0.1	0.499%
0.2	1.000%
0.3	1.513%
0.4	2.048%
0.5	2.612%
0.6	3.215%
0.7	3.864%
0.8	4.567%
0.9	5.331%
1.0	6.163%
1.1	7.068%
1.2	8.051%
1.3	9.118%
1.4	10.273%
1.5	11.518%
1.6	12.856%
1.7	14.289%
1.8	15.817%
1.9	17.442%
2.0	19.161%
2.1	20.974%
2.2	22.877%
2.3	24.868%
2.4	26.941%
2.5	29.093%

大域誤差率
10.8137442409684 %

このことから, x が大きくなるにつれて誤差率が大きくなっていることがわかる.
オイラー法による局所誤差は一般に $O(h^2)$ であり, これは計算した誤差率とほとんど一致する.
また, 大域誤差は一般に $O(h)$ であり, これも計算した誤差率とほとんど一致する.

課題 3.2

3.2.a 問題:

次の 2 階微分方程式をオイラー法で解け. ただし, $h = 0.4, 0 \leq x \leq 6$ とせよ.

$$\frac{d^2y}{dx^2} = e^x - y - \frac{dy}{dx} \text{初期条件 } x = 0 \text{ で } y = 1, \frac{dy}{dx} = 1 \tag{2}$$

得られた結果を解析解：

$$y = \frac{2}{3}e^{-\frac{x}{2}}(\cos\frac{\sqrt{3}}{2}x + \sqrt{3}\sin\frac{\sqrt{3}}{2}x) + \frac{1}{3}e^{x} \tag{3}$$

とともにグラフで表し比較せよ.

3.2.b アルゴリズム:

2 階微分方程式におけるオイラー法のアルゴリズムを図 3 に示す.

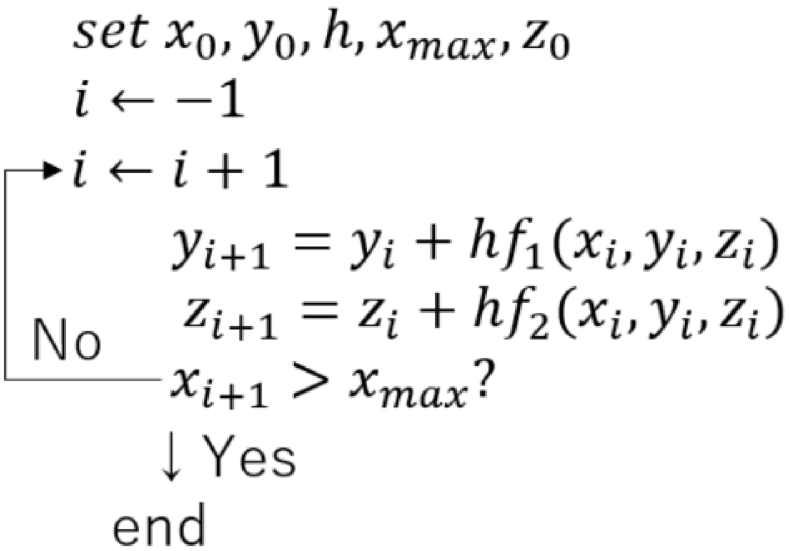

図 3　3.2 2 階微分方程式におけるオイラー法のアルゴリズム

3.2.c プログラムリスト:

使用したプログラムを以下に示す.

ソースコード 2　演習課題 3.2 Python プログラム

```
#3-2
import numpy as np
import math
import sympy as sym
import matplotlib.pyplot as plt
import japanize_matplotlib

#オイラー法
def EulerMethod(x_val, y_val, z_val, h, x_max, f1, f2, x, y, z):
    i = 0
```

```
    while(True):
        y_val.append(y_val[i] + h * f1.subs([(x, x_val[i]), (y, y_val[i]), (z, z_val[i])]))
        z_val.append(z_val[i] + h * f2.subs([(x, x_val[i]), (y, y_val[i]), (z, z_val[i])]))
        x_val.append(x_val[i] + h)
        if(x_val[i+1] > x_max):
            break;
        i += 1

def error(est, act): #誤差率を求める関数
    return abs(100 * (est - act) / act)

def main():
    x_val = [0]
    y_val = [1]
    z_val = [1]
    h = 0.4
    x_max = 6
    x = sym.symbols("x")
    y = sym.symbols("y")
    z = sym.symbols("z")
    f1 = z
    f2 = math.e ** x - y - z
    EulerMethod(x_val, y_val, z_val, h, x_max, f1, f2, x, y, z)

    x2 = np.linspace(0, 7, 100)
    fx = (2/3)*np.exp(-x2/2) * (np.cos(np.sqrt(3)/2*x2) + np.sqrt(3)*np.sin(np.sqrt(3)/2*x2)) + (1/3)*np.exp(x2)

    plt.plot(x2, fx, label = "与えられた解析解")
    plt.xlim(0, 6)
    plt.ylim(0, 120)
    #近似曲線
    plt.plot(x_val, y_val, 'r--', label = '得られた結果')

    plt.xlabel("x")
    plt.ylabel("y")

    plt.legend()
    plt.show()

```

```
    #誤差率
    px = np.linspace(0, x_max, int(x_max/h+1))
    py = (2/3)*np.exp(-px/2) * (np.cos(np.sqrt(3)/2*px) + np.sqrt(3)*np.sin(np.
        sqrt(3)/2*px)) + (1/3)*np.exp(px)
    print(" x | 誤差率 ")
    print("-------------")
    for i in range(len(px)):
        print(f"{px[i]:.1f}", f"{round(error(py[i], y_val[i]), 3):6.3f}", end = "%
            \n", sep = " | ")

if __name__ == "__main__":
    main()
```

3.2.d 実行結果:

実行結果を図 4 に示す.

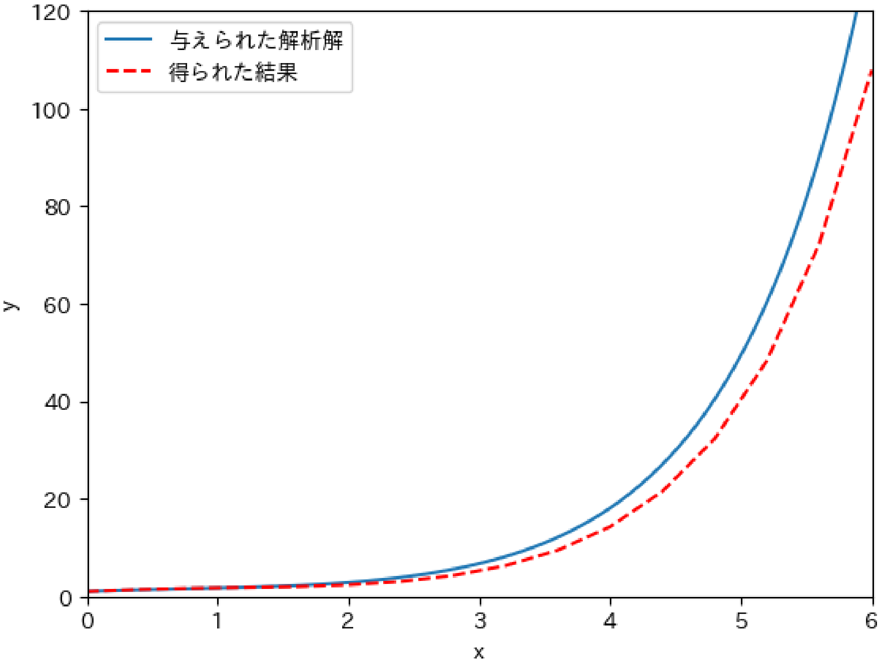

図 4　3.2 実行結果

3.2.e 考察:

図 4 から得られた解析解と与えられた解析解は似たような曲線になっていることが分かる.
また出力した局所誤差を以下に示す.

```
局所誤差
   x | 誤差率
-------------
0.0 |   0.000%
0.4 |   5.132%
0.8 |   3.793%
1.2 |   2.162%
1.6 |  10.312%
2.0 |  17.537%
2.4 |  21.847%
2.8 |  23.293%
3.2 |  23.016%
3.6 |  22.102%
4.0 |  21.182%
4.4 |  20.495%
4.8 |  20.071%
5.2 |  19.851%
5.6 |  19.762%
6.0 |  19.743%

大域誤差率
15.6435321350716 %
```

局所誤差を見ると $x = 0.4$ 3.2 にかけて誤差率が上昇しているが, $x = 3.2$ を超えると誤差率が 20%前後で安定していることが分かる.
また, 大域誤差は 15.64%で, オイラー法の一般的な大域誤差 $O(h)$ とほとんど一致している.

課題 3.3

3.3.a 問題:

次の微分方程式の境界値問題を解け. ただし, 分割数 $n = 10$ とせよ.

$$\frac{d^2y}{dx^2} - 4y = 0, \text{境界条件} : x = 0 \text{ で } y = 0, x = 1 \text{ で } y = 0.25 \tag{4}$$

得られた結果を解析解 :

$$y = \frac{1}{4}\frac{e^{2x} - e^{-2x}}{e^2 - e^{-2}} \tag{5}$$

とともにグラフで表し比較せよ.

3.3.b アルゴリズム:

2 点以上の点 x に対する y または y の微分値が得られている問題を境界値問題という. 次の 2 階微分方程式について考える.

$$\frac{d^2}{dx^2} + A(x)\frac{d}{dy}y + B(x)y = C(x)(1) \tag{6}$$

境界条件：$x=a$ の時, $y=c$ であり, $x=b$ の時 $y=d$
この時境界条件は図 5 のようになる. この解法は差分方程式に置き換えて, 連立 1 次方程式を解く問題に帰着させる.

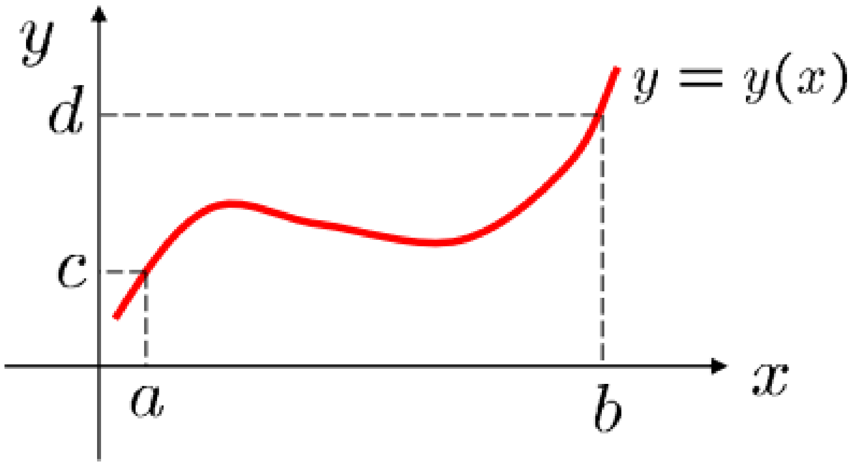

図 5　3.3 境界条件

まず, $y_i = y(x_i), y_{i+1} = y(x_i + h), h$: 刻み幅とする. テイラー展開より,

$$y_{i+1} = y_i + hy_i^{'} + h^2\frac{y_i^{''}}{2!} + h^3\frac{y_i^{(3)}}{3!} + \cdots \tag{7}$$

また, 式の h を $-h$ とすると,

$$y_{i+1} = y_i - hy_i^{'} + h^2\frac{y_i^{''}}{2!} - h^3\frac{y_i^{(3)}}{3!} + \cdots \tag{8}$$

h^3以上の項は無視できるほど小さいものと仮定すると式から

$$y_i^{'} = \frac{1}{2h}(y_{i+1} - y_{i-1}) \tag{9}$$

同様にして式から h^3以上の項を無視すると

$$y_i^{''} = \frac{1}{h^2}(y_{i+1} - 2y_i + y_{i-1}) \tag{10}$$

と式の近似式を式に代入すると, 次の差分方程式が得られる.

$$\frac{1}{h^2}(y_{i+1} - 2y_i + y_{i-1}) + A(x_i)\frac{1}{2h}(y_{i+1} - y_{i-1}) + B(x_i)y_i = C(x_i) \tag{11}$$

この式は $i = 1, 2, \cdots, n-1$ について成立するので未知数 $\{y_1, y_2 \cdots, y_{n-1}\}$ を持つ $(n-1)$ 個の連立 1 次方程式に変換される. この連立方程式の係数行列は, 対角要素付近以外はすべて 0 となるので扱いやすい.

したがって今回の問題は次のような行列を逆行列法を用いて解く.

$$
\begin{pmatrix}
-2.04 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -2.04 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -2.04 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -2.04 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -2.04 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -2.04 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -2.04 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -2.04 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2.04
\end{pmatrix}
\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{pmatrix}
=
\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -0.25 \end{pmatrix}
$$

また, 逆行列法のアルゴリズムを図 6 に示す.

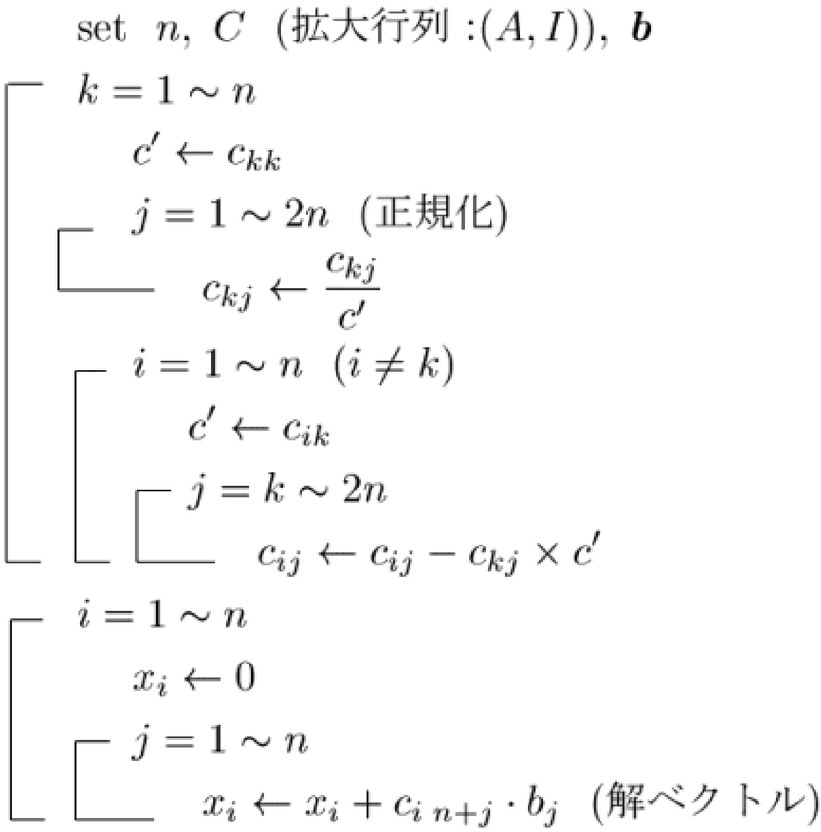

図 6　3.3 逆行列法のアルゴリズム

3.3.c プログラムリスト:

使用したプログラムを以下に示す.

ソースコード 3　演習課題 3.3 Python プログラム

```
#3-3
import numpy as np
import math
import sympy as sym
import matplotlib.pyplot as plt
import japanize_matplotlib

def error(est, act): #誤差率を求める関数
    return abs(100 * (est - act) / act)

def main():
    y_min = 0
    x_min = 0
    x_max = 1
    y_max = 0.25

    x_val = np.linspace(0, 1, 11)
    y_val = [y_min]

    n = 9
    h = (x_max - x_min) / (n+1) * 2

    y = np.array([1]*n, dtype = 'float')

    C = np.zeros((n, n))
    for i in range(0, n):
        for j in range(0, n):
            if i == j:
                C[j, i] = -2.04
            elif i == j - 1:
                C[j, i] = 1
            elif i == j + 1:
                C[j, i] = 1

    unit = np.identity(n)
    b=np.array([0, 0, 0, 0, 0, 0, 0, 0, -0.25])

    C = np.hstack([C, unit])

    #逆行列法
    for k in range(n):
```

```
        Ctmp = C[k][k]
        for j in range(2 * n):
            C[k][j] /= Ctmp #正規化
        for i in range(n):
            if i != k:
                Ctmp = C[i][k]
                for j in range(k, 2 * n):
                    C[i][j] = C[i][j] - C[k][j] * Ctmp

    for i in range(n):
        y[i] = 0
        for j in range(n):
            y[i] = y[i] + C[i][n+j] * b[j]
        y_val.append(y[i])

    y_val.append(y_max)

    x = sym.symbols(’x’)
    expr = (1/4)*( (sym.exp(2*x)-sym.exp(-2*x)) / (sym.exp(2)-sym.exp(-2)) )
    expr_func = sym.lambdify(x, expr, "numpy")
    px=np.linspace(0, 1, 100)
    py=expr_func(px)
    plt.plot(px, py, color = "blue", label = ’与えられた解析解’)
    plt.plot(x_val, y_val, ’ro--’, label = r’得られた結果’)

    plt.xlim(x_min, x_max)
    plt.xlabel("x")
    plt.ylabel("y")

    plt.legend()
    plt.show()

    #誤差率
    py = expr_func(x_val)
    print(" x | 誤差率 ")
    print("-------------")
    for i in range(len(py)):
        print(f"{x_val[i]:.1f}", f"{round(error(py[i], y_val[i]), 3):6.3f}", end = "% \n", sep = " | ")

if __name__ == "__main__":
    main()
```

3.3.d 実行結果:

実行結果を図 7 に示す.

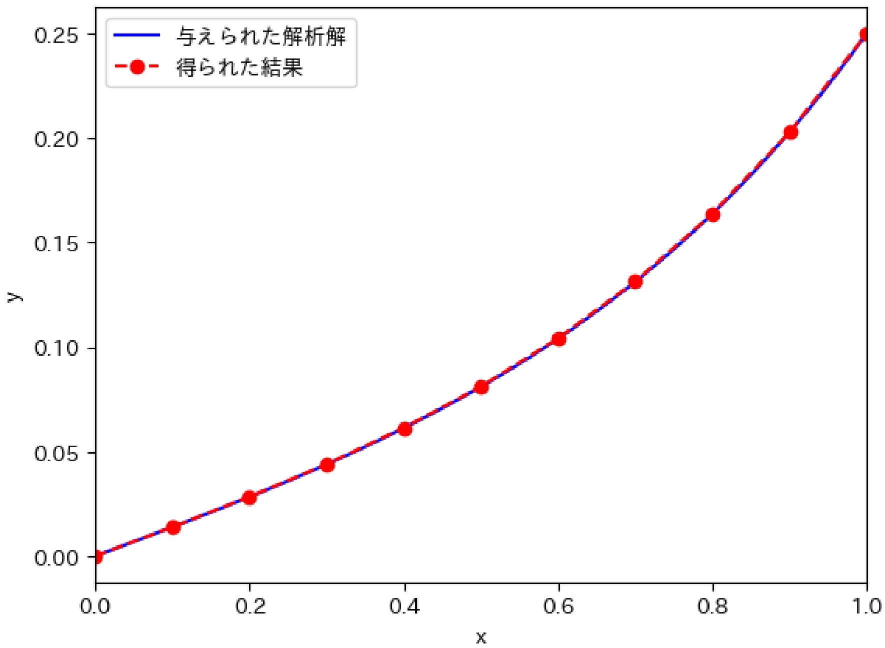

図 7　3.3 実行結果

3.3.e 考察:

図 7 から得られた解析解と与えられた解析解はほとんど一致していることがわかる. また出力した誤差を以下に示す.

```
局所誤差
  x | 誤差率
-------------
0.0 |   0.000%
0.1 |   0.176%
0.2 |   0.170%
0.3 |   0.159%
0.4 |   0.144%
0.5 |   0.126%
0.6 |   0.105%
0.7 |   0.082%
0.8 |   0.056%
0.9 |   0.029%
1.0 |   0.000%

大域誤差率
0.09523961693138934 %
```

局所誤差率も大域誤差率もどちらもかなり小さくなっている.
これは逆行列法を使用しており, 逆行列法は正確な値がでるアルゴリズムだからである.

課題 3.4

3.4.a 問題:

次の 1 階微分方程程式をオイラー法, 2 次ルンゲクッタ法, 3 次のルンゲクッタ方と 4 次のルンゲクッタ法で解け, ただし, $h = 0.5, 1 \leq x \leq 20$ とせよ.

$$\frac{dy}{dx} = 2\frac{y}{x}, \text{初期条件 } x = 1 \text{ で } y = 1 \tag{12}$$

得られた結果を解析解 $y = x^2$とともにグラフで表し比較せよ.

3.4.b アルゴリズム:

オイラー法のアルゴリズムは 3.1.b で示した通りである.
2 次ルンゲクッタ法, 3 次ルンゲクッタ法, 4 次ルンゲクッタ法のアルゴリズムをそれぞれ図 8 10 に示す.

$$\text{初期条件 }:h, x_0, y_0$$
$$y_{i+1} = y_i + hf(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}) = y_i + k_2$$
$$\text{ただし}, k_1 = hf(x_i, y_i),\ k_2 = hf(x_i + \frac{h}{2}, y_i + \frac{k_1}{2})$$

図 8　3.4 2 次ルンゲクッタ法アルゴリズム

$$
\begin{aligned}
&\text{初期条件 }: h, x_0, y_0\\
&y_{i+1} = y_i + \frac{k_1 + 4k_2 + k_3}{6} \qquad (\\
&\text{ただし},\\
&k_1 = hf(x_i, y_i),\\
&k_2 = hf(x_i + \frac{h}{2}, y_i + \frac{k_1}{2})\\
&k_3 = hf(x_i + h, y_i + 2k_2 - k_1)
\end{aligned}
$$

図 9　3.4 3 次ルンゲクッタ法アルゴリズム

$$
\begin{aligned}
&\text{初期条件 }: h, x_0, y_0\\
&y_{i+1} = y_i + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}\\
&\text{ただし},\\
&k_1 = hf(x_i, y_i),\\
&k_2 = hf(x_i + \frac{h}{2}, y_i + \frac{k_1}{2})\\
&k_3 = hf(x_i + \frac{h}{2}, y_i + \frac{k_2}{2})\\
&k_4 = hf(x_i + h, y_i + k_3)
\end{aligned}
$$

図 10　3.4 4 次ルンゲクッタ法アルゴリズム

3.4.c プログラムリスト:

使用したプログラムを以下に示す.

ソースコード 4　演習課題 3.4 Python プログラム

```
#3-4
import numpy as np
import math
import sympy as sym
import matplotlib.pyplot as plt
import japanize_matplotlib

#オイラー法
def EulerMethod(x_val, y_val, h, x_max, f, x, y):
    i = 0
    while(True):
        if(x_val[i] >= x_max):
            break

        y_val.append(y_val[i] + h * f.subs([(x, x_val[i]), (y, y_val[i])]))
        x_val.append(x_val[i] + h)

        i += 1

#2次ルンゲクッタ
def RungeKuttaMethod2(x_val, y_val, h, x_max, f, x, y):
    i = 0
    while(True):
        k1 = h * f.subs([(x, x_val[i]), (y, y_val[i])])
        k2 = h * f.subs([(x, x_val[i] + h/2), (y, y_val[i] + k1/2)])

        if(x_val[i] >= x_max):
            break

        y_val.append(y_val[i] + k2)
        x_val.append(x_val[i] + h)

        i += 1

#3次ルンゲクッタ
def RungeKuttaMethod3(x_val, y_val, h, x_max, f, x, y):
    i = 0
```

```
    while(True):
        k1 = h * f.subs([(x, x_val[i]), (y, y_val[i])])
        k2 = h * f.subs([(x, x_val[i] + h/2), (y, y_val[i] + k1/2)])
        k3 = h * f.subs([(x, x_val[i] + h), (y, y_val[i] + 2*k2 - k1)])

        if(x_val[i] >= x_max):
            break

        y_val.append(y_val[i] + (k1 + 4*k2 + k3)/6)
        x_val.append(x_val[i] + h)

        i += 1

#4次ルンゲクッタ
def RungeKuttaMethod4(x_val, y_val, h, x_max, f, x, y):
    i = 0
    while(True):
        k1 = h * f.subs([(x, x_val[i]), (y, y_val[i])])
        k2 = h * f.subs([(x, x_val[i] + h/2), (y, y_val[i] + k1/2)])
        k3 = h * f.subs([(x, x_val[i] + h/2), (y, y_val[i] + k2/2)])
        k4 = h * f.subs([(x, x_val[i] + h), (y, y_val[i] + k3)])

        if(x_val[i] >= x_max):
            break

        y_val.append(y_val[i] + (k1 + 2*k2 + 2*k3 + k4)/6)
        x_val.append(x_val[i] + h)

        i += 1

def error(est, act): #誤差率を求める関数
    return abs(100 * (est - act) / act)

def main():
    x_val = [[1], [1], [1], [1]]
    y_val = [[1], [1], [1], [1]]
    h = 0.5
    x_max = 20
    x_min = 1
    x = sym.symbols("x")
    y = sym.symbols("y")
    f = 2 * (y / x)
```

```
    EulerMethod(x_val[0], y_val[0], h, x_max, f, x, y)
    RungeKuttaMethod2(x_val[1], y_val[1], h, x_max, f, x, y)
    RungeKuttaMethod3(x_val[2], y_val[2], h, x_max, f, x, y)
    RungeKuttaMethod4(x_val[3], y_val[3], h, x_max, f, x, y)

    #グラフ
    x = sym.symbols('x')
    expr = x**2
    expr_func = sym.lambdify(x, expr, "numpy")
    px = np.linspace(x_min, x_max, 100)
    py = expr_func(px)

    plt.plot(px, py, color = "blue", label = '与えられた解析解')
    plt.plot(x_val[0], y_val[0], label = 'オイラー法')
    plt.plot(x_val[1], y_val[1], label = '2次ルンゲクッタ')
    plt.plot(x_val[2], y_val[2], label = '3次ルンゲクッタ')
    plt.plot(x_val[3], y_val[3], label = '4次ルンゲクッタ')

    plt.xlim(x_min, x_max)
    plt.xlabel("x")
    plt.ylabel("y")

    plt.legend()
    plt.show()

    #局所誤差
    px = np.linspace(x_min, x_max, int(x_max/h-1))
    py = expr_func(px)
    print("解析解と各アルゴリズムとの局所誤差: ")
    print(" x | オイラー法 | 2次ルンゲクッタ | 3次ルンゲクッタ | 4次ルンゲクッタ |")
    print("-------------------------------------------------------------------------------------")

    for i in range(len(py)):
        print(f"{px[i]: <5.1f}", end = " | ")
        for j in range(len(y_val)):
            print(f"{error(py[i], y_val[j][i]):<14.6f}", end = "% | ")
        print()

if __name__ == "__main__":
    main()
```

3.4.d 実行結果:

実行結果を図 11 に示す. また, x の範囲を $18 \leq x \leq 19$, y の範囲を $320 \leq y \leq 350$ に拡大した時の実行結果を図 12 に示す.

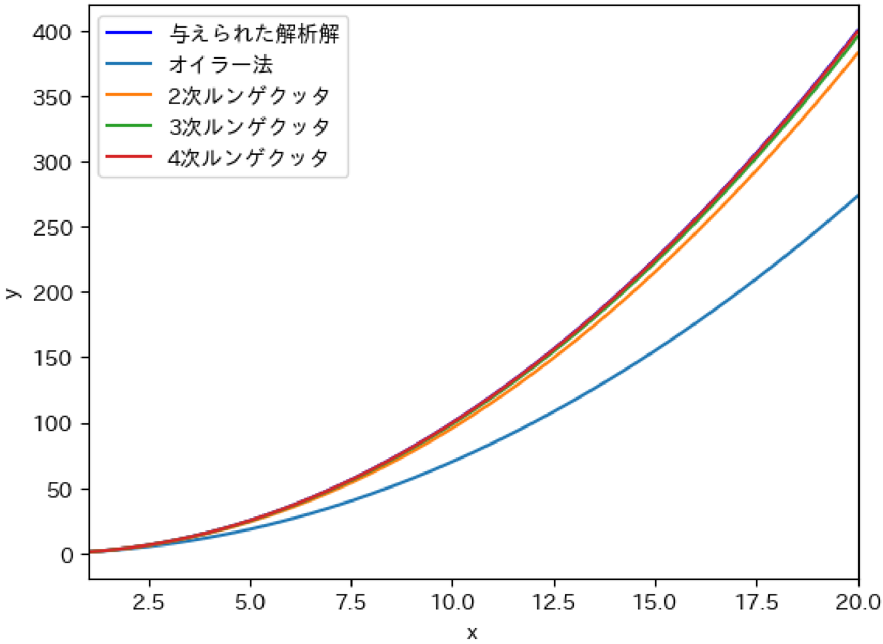

図 11　3.4 実行結果

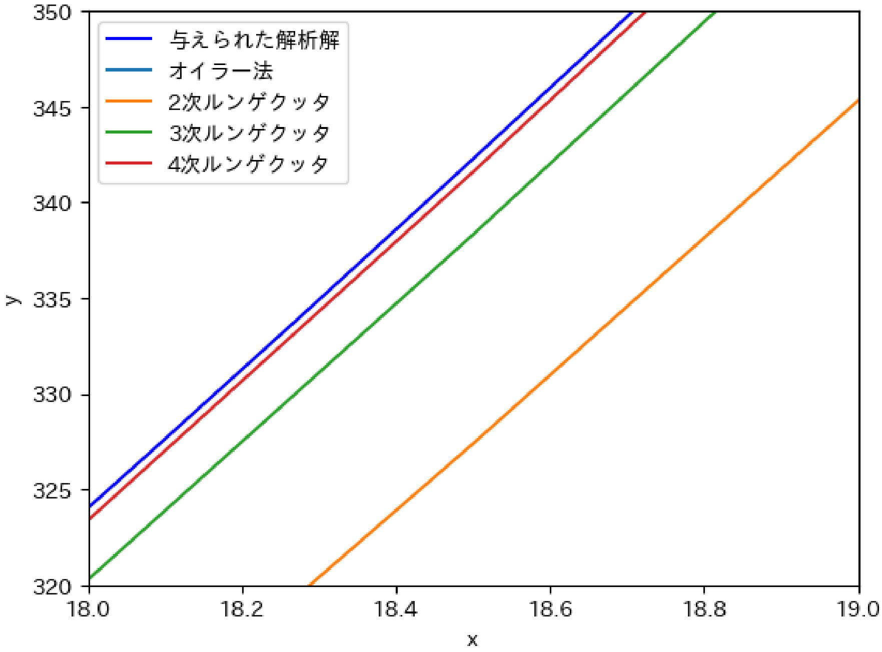

図 12　3.4 実行結果

3.4.e 考察:

図 11, 12 を見ると解析解の曲線と比べて 4 次 $\geq$ 3 次 $\geq$ 2 次 $\geq$ オイラー法という順に近似していることが分かる.
また, 出力した局所誤差を以下に示す.

解析解と各アルゴリズムとの局所誤差:

x	オイラー法	2次ルンゲクッタ	3次ルンゲクッタ	4次ルンゲクッタ
1.0	0.000000 %	0.000000 %	0.000000 %	0.000000 %
1.5	11.111111 %	2.222222 %	0.740741 %	0.148148 %
2.0	16.666667 %	3.095238 %	0.962302 %	0.179989 %
2.5	20.000000 %	3.525926 %	1.050335 %	0.189847 %
3.0	22.222222 %	3.769547 %	1.091981 %	0.193666 %
3.5	23.809524 %	3.920616 %	1.114162 %	0.195388 %
4.0	25.000000 %	4.020698 %	1.127038 %	0.196254 %
4.5	25.925926 %	4.090400 %	1.135016 %	0.196728 %
5.0	26.666667 %	4.140879 %	1.140220 %	0.197005 %
5.5	27.272727 %	4.178604 %	1.143756 %	0.197175 %
6.0	27.777778 %	4.207535 %	1.146244 %	0.197284 %
6.5	28.205128 %	4.230208 %	1.148044 %	0.197357 %
7.0	28.571429 %	4.248305 %	1.149378 %	0.197406 %
7.5	28.888889 %	4.262980 %	1.150388 %	0.197442 %
8.0	29.166667 %	4.275043 %	1.151166 %	0.197467 %
8.5	29.411765 %	4.285081 %	1.151776 %	0.197486 %
9.0	29.629630 %	4.293521 %	1.152260 %	0.197500 %
9.5	29.824561 %	4.300686 %	1.152650 %	0.197510 %
10.0	30.000000 %	4.306821 %	1.152967 %	0.197518 %
10.5	30.158730 %	4.312113 %	1.153227 %	0.197525 %
11.0	30.303030 %	4.316711 %	1.153443 %	0.197530 %
11.5	30.434783 %	4.320731 %	1.153623 %	0.197534 %
12.0	30.555556 %	4.324265 %	1.153775 %	0.197537 %
12.5	30.666667 %	4.327389 %	1.153905 %	0.197540 %
13.0	30.769231 %	4.330164 %	1.154015 %	0.197542 %
13.5	30.864198 %	4.332640 %	1.154110 %	0.197544 %
14.0	30.952381 %	4.334859 %	1.154191 %	0.197545 %
14.5	31.034483 %	4.336854 %	1.154263 %	0.197547 %
15.0	31.111111 %	4.338656 %	1.154325 %	0.197548 %
15.5	31.182796 %	4.340288 %	1.154379 %	0.197549 %
16.0	31.250000 %	4.341771 %	1.154427 %	0.197549 %
16.5	31.313131 %	4.343122 %	1.154469 %	0.197550 %
17.0	31.372549 %	4.344357 %	1.154507 %	0.197551 %
17.5	31.428571 %	4.345489 %	1.154540 %	0.197551 %
18.0	31.481481 %	4.346528 %	1.154570 %	0.197551 %
18.5	31.531532 %	4.347485 %	1.154597 %	0.197552 %
19.0	31.578947 %	4.348369 %	1.154621 %	0.197552 %
19.5	31.623932 %	4.349185 %	1.154642 %	0.197552 %
20.0	31.666667 %	4.349942 %	1.154662 %	0.197553 %

オイラー法の局所誤差は $O(h^2)$, 2 次ルンゲクッタ法は $O(h^3)$, 3 次ルンゲクッタ法は $O(h^4)$, 4 次ルンゲクッタ法は $O(h^5)$ であり, 出力した局所誤差と比べるとほとんど一致していることがわかる.

課題 3.5

3.5.a 問題:

次の微分方程式の境界値問題を解け. ただし, 分割数 $n = 10$ とせよ.

$$\frac{d^2y}{dx^2} = x + y, \text{初期条件 } x = 0 \text{ で } y = 0, x = 1 \text{ で } y = 1 \tag{13}$$

得られた結果を解析解 :

$$y = \frac{1}{2}(e^x - e - x) - x + \frac{e^x - e^{-x}}{e - e^{-1}}(\frac{1}{2}(e^{-1} - e) + 2) \tag{14}$$

とともにグラフで表し比較せよ.

3.5.b アルゴリズム:

アルゴリズムは $3.3.b$ に示した通りで, この問題では以下のような行列式を逆行列法を用いて解けばよい.

$$
\begin{pmatrix}
-2.01 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -2.01 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -2.01 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -2.01 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -2.01 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -2.01 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -2.01 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -2.01 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2.01
\end{pmatrix}
\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{pmatrix}
=
\begin{pmatrix} 0.001 \\ 0.002 \\ 0.003 \\ 0.004 \\ 0.005 \\ 0.006 \\ 0.007 \\ 0.008 \\ -0.991 \end{pmatrix}
$$

3.5.c プログラムリスト:

使用したプログラムを以下に示す.

ソースコード 5　演習課題 3.5 Python プログラム

```
#3-5
import numpy as np
import math
import sympy as sym
import matplotlib.pyplot as plt
import japanize_matplotlib

def error(est, act): #誤差率を求める関数
    return abs(100 * (est - act) / act)

def main():
    y_min = 0
    x_min = 0
    x_max = 1
    y_max = 1

    x_val = np.linspace(0, 1, 11)
    y_val = [y_min]

    n = 9
    h = (x_max - x_min) / (n + 1)

```

```
y = np.array([1]*n, dtype = 'float')

C = np.zeros((n, n))
for i in range(0, n):
    for j in range(0, n):
        if i == j:
            C[j, i] = -2.01
        elif i == j - 1:
            C[j, i] = 1
        elif i == j + 1:
            C[j, i] = 1

unit = np.identity(n)
b = [0.001, 0.002, 0.003, 0.004, 0.005, 0.006, 0.007, 0.008, -0.991]

C = np.hstack([C, unit])

for k in range(n):
    Ctmp = C[k][k]
    for j in range(2 * n):
        C[k][j] /= Ctmp #正規化
    for i in range(n):
        if i != k:
            Ctmp = C[i][k]
            for j in range(k, 2 * n):
                C[i][j] = C[i][j] - C[k][j] * Ctmp

for i in range(n):
    y[i] = 0
    for j in range(n):
        y[i] = y[i] + C[i][n+j] * b[j]
    y_val.append(y[i])

y_val.append(y_max)

x = sym.symbols('x')
expr = (1/2) * (sym.exp(x)-sym.exp(-x)) -x + ((sym.exp(x)-sym.exp(-x)) / (sym.
    exp(1)-sym.exp(-1))) *((1/2)*(sym.exp(-1)-sym.exp(1))+2)
expr_func = sym.lambdify(x, expr, "numpy")
px=np.linspace(0, 1, 100)
py=expr_func(px)
plt.plot(px, py, color = "blue", label = r'与えられた解析解')
```

```
    plt.plot(x_val, y_val, 'ro--', label = r'得られた解析解')

    plt.xlim(x_min, x_max)
    plt.xlabel("x")
    plt.ylabel("y")

    plt.legend()
    plt.show()

    #誤差率
    px = np.linspace(x_min, x_max, int(x_max/h+1))
    py = expr_func(px)

    print(" x | 誤差率 ")
    print("-------------")
    for i in range(len(py)):
        print(f"{px[i]:.1f}", f"{round(error(py[i], y_val[i]), 4):6.4f}", end = "%\n", sep = " | ")

if __name__ == "__main__":
    main()
```

3.5.d 実行結果:

実行結果を図 13 に示す.

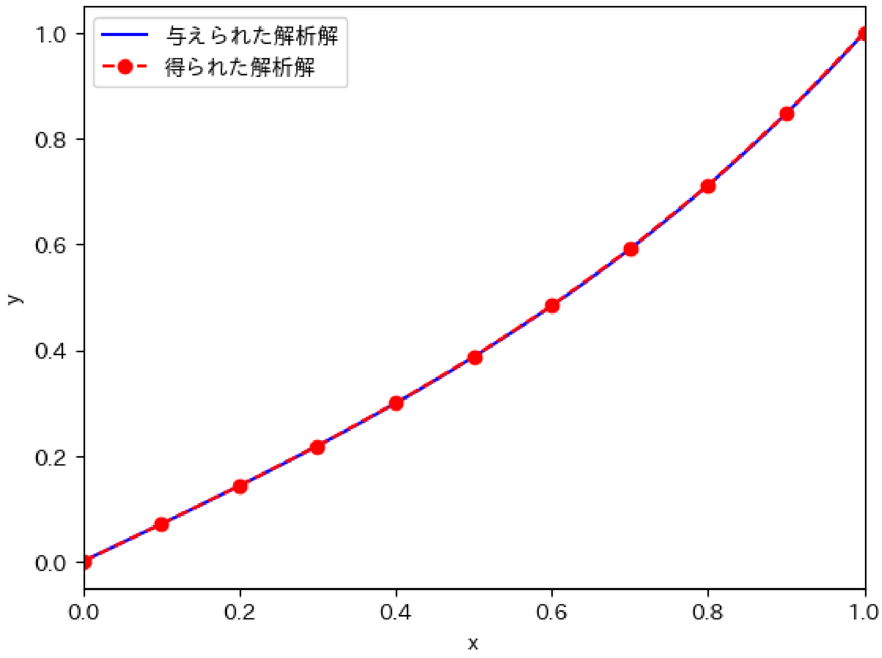

図 13　3.5 実行結果

3.5.e 考察:

3.3 どうように逆行列法を用いて解析解を求めているため, 与えられた解析解と正確に近似していることが分かる.
また, 出力した局所誤差を以下に示す.

x	誤差率
0.0	0.0000%
0.1	0.0312%
0.2	0.0300%
0.3	0.0280%
0.4	0.0253%
0.5	0.0220%
0.6	0.0183%
0.7	0.0141%
0.8	0.0096%
0.9	0.0049%
1.0	0.0000%

出力した局所誤差を見ても, 正確な値が求められていることが分かる.

課題 3.6

3.6.a 問題:

次の 2 階微分方程式を 2 次のルンゲクッタ法と 4 次のルンゲクッタ法で解け. ただし, $h = 0.5, 0 \leq x \leq 5$ とせよ.

$$\frac{d^2y}{dx^2} + \frac{dy}{dx} + y = e^x, \text{初期条件 } x = 0 \text{ で } y = 1, \frac{dy}{dx} = 1 \tag{15}$$

得られた結果を解析解：

$$y = \frac{2}{3}e^{-\frac{x}{2}}(\cos\frac{\sqrt{3}}{2}x + \sqrt{3}\sin\frac{\sqrt{3}}{2}x) + \frac{e^x}{3} \tag{16}$$

とともにグラフで表し比較せよ.

3.6.b アルゴリズム:

$z = \frac{dy}{dx}$ と置くと, $\frac{dz}{dx} = \frac{d^2y}{dx^2}$ となり, 問題の式から

$$\begin{cases} \dfrac{dy}{dx} = z \\ \dfrac{dz}{dx} = e^x - y - z \end{cases} \tag{17}$$

という 2 元連立 1 階常微分方程式が得られる.
2 元連立 1 階微分方程式の 2 次ルンゲクッタ法での解法を図 14, 4 次ルンゲクッタ法での解法を図 15 に示す.

初期値条件 $:h, x_0, y_0, z_0$

$$\begin{cases} k_{y1} = hf_1(x_i, y_i, z_i) \\ k_{z1} = hf_2(x_i, y_i, z_i) \\ k_{y2} = hf_1(x_i + \frac{h}{2}, y_i + \frac{k_{y1}}{2}, z_i + \frac{k_{z1}}{2}) \\ k_{z2} = hf_2(x_i + \frac{h}{2}, y_i + \frac{k_{y1}}{2}, z_i + \frac{k_{z1}}{2}) \end{cases}$$

$$\begin{cases} y_{i+1} = y_i + k_{y2} \\ z_{i+1} = z_i + k_{z2} \end{cases}$$

$$x_{i+1} = x_i + h$$

図 14　2 元連立 1 階微分方程式の 2 次ルンゲクッタ法での解法

$$初期条件\ :h, x_0, y_0, z_0$$

$$\begin{cases} k_{y1} = hf_1(x_i, y_i, z_i) \\ k_{z1} = hf_2(x_i, y_i, z_i) \\ k_{y2} = hf_1(x_i + \frac{h}{2}, y_i + \frac{k_{y1}}{2}, z_i + \frac{k_{z1}}{2}) \\ k_{z2} = hf_2(x_i + \frac{h}{2}, y_i + \frac{k_{y1}}{2}, z_i + \frac{k_{z1}}{2}) \\ k_{y3} = hf_1(x_i + \frac{h}{2}, y_i + \frac{k_{y2}}{2}, z_i + \frac{k_{z2}}{2}) \\ k_{z3} = hf_2(x_i + \frac{h}{2}, y_i + \frac{k_{y2}}{2}, z_i + \frac{k_{z2}}{2}) \\ k_{y4} = hf_1(x_i + h, y_i + k_{y3}, z_i + k_{z3}) \\ k_{z4} = hf_2(x_i + h, y_i + k_{y3}, z_i + k_{z3}) \end{cases}$$

$$\begin{cases} y_{i+1} = y_i + \frac{1}{6}(k_{y1} + 2k_{y2} + 2k_{y3} + k_{y4}) \\ z_{i+1} = z_i + \frac{1}{6}(k_{z1} + 2k_{z2} + 2k_{z3} + k_{z4}) \end{cases}$$

$$x_{i+1} = x_i + h$$

図 15　2 元連立 1 階微分方程式の 4 次ルンゲクッタ法での解法

3.6.c プログラムリスト:

使用したプログラムを以下に示す.

ソースコード 6　演習課題 3.6 Python プログラム

```
#3-6
import numpy as np
import math
import sympy as sym
import matplotlib.pyplot as plt
import japanize_matplotlib
```

```

#2次ルンゲクッタ
def RungeKuttaMethod2(x_val, y_val, z_val, h, x_max, f1, f2, x, y, z):
    i = 0
    while(True):
        ky1 = h * f1.subs([(x, x_val[i]), (y, y_val[i]), (z, z_val[i])])
        kz1 = h * f2.subs([(x, x_val[i]), (y, y_val[i]), (z, z_val[i])])
        ky2 = h * f1.subs([(x, x_val[i] + h/2), (y, y_val[i] + ky1/2), (z, z_val[i] + kz1/2)])
        kz2 = h * f2.subs([(x, x_val[i] + h/2), (y, y_val[i] + ky1/2), (z, z_val[i] + kz1/2)])

        if(x_val[i] >= x_max):
            break

        x_val.append(x_val[i] + h)
        y_val.append(y_val[i] + ky2)
        z_val.append(z_val[i] + kz2)

        i += 1

#4次ルンゲクッタ
def RungeKuttaMethod4(x_val, y_val, z_val, h, x_max, f1, f2, x, y, z):
    i = 0
    while(True):
        ky1 = h * f1.subs([(x, x_val[i]), (y, y_val[i]), (z, z_val[i])])
        kz1 = h * f2.subs([(x, x_val[i]), (y, y_val[i]), (z, z_val[i])])
        ky2 = h * f1.subs([(x, x_val[i] + h/2), (y, y_val[i] + ky1/2), (z, z_val[i] + kz1/2)])
        kz2 = h * f2.subs([(x, x_val[i] + h/2), (y, y_val[i] + ky1/2), (z, z_val[i] + kz1/2)])
        ky3 = h * f1.subs([(x, x_val[i] + h/2), (y, y_val[i] + ky2/2), (z, z_val[i] + kz2/2)])
        kz3 = h * f2.subs([(x, x_val[i] + h/2), (y, y_val[i] + ky2/2), (z, z_val[i] + kz2/2)])
        ky4 = h * f1.subs([(x, x_val[i] + h), (y, y_val[i] + ky2), (z, z_val[i] + kz3)])
        kz4 = h * f2.subs([(x, x_val[i] + h), (y, y_val[i] + ky3), (z, z_val[i] + kz3)])

        if(x_val[i] >= x_max):
            break
```

```
        x_val.append(x_val[i] + h)
        y_val.append(y_val[i] + (1/6) * (ky1 + 2*ky2 + 2*ky3 + ky4))
        z_val.append(z_val[i] + (1/6) * (kz1 + 2*kz2 + 2*kz3 + kz4))

        i += 1

def error(est, act): #誤差率を求める関数
    return abs(100 * (est - act) / act)

def main():
    x_val = [[0], [0]]
    y_val = [[1], [1]]
    z_val = [[1], [1]]
    h = 0.5
    x_max = 5
    x_min = 0
    x = sym.symbols("x")
    y = sym.symbols("y")
    z = sym.symbols("z")
    f1 = z
    f2 = sym.exp(x) - y - z
    RungeKuttaMethod2(x_val[0], y_val[0], z_val[0], h, x_max, f1, f2, x, y, z)
    RungeKuttaMethod4(x_val[1], y_val[1], z_val[0], h, x_max, f1, f2, x, y, z)

    #グラフ
    x=sym.symbols('x')
    expr=(2/3) * (sym.exp(-x/2)) * ( sym.cos((sym.sqrt(3)*x)/2) + sym.sqrt(3)*sym
        .sin((sym.sqrt(3)*x)/2) ) + sym.exp(x)/3
    expr_func=sym.lambdify(x,expr,"numpy")
    px=np.linspace(0,5,100)
    py=expr_func(px)

    plt.plot(px, py, color = "blue", label = '与えられた解析解')
    plt.plot(x_val[0], y_val[0], label = '2次ルンゲクッタ')
    plt.plot(x_val[1], y_val[1], label = '4次ルンゲクッタ')

    plt.xlim(0, 5)
    plt.xlabel("x")
    plt.ylabel("y")

    #近似曲線
```

```
    plt.legend()
    plt.show()

    #局所誤差
    px = np.linspace(x_min, x_max, int(x_max/h+1))
    py = expr_func(px)
    print("解析解と各アルゴリズムとの局所誤差: ")
    print(" x | 2次ルンゲクッタ | 4次ルンゲクッタ |")
    print("-----------------------------------------")
    for i in range(len(py)):
        print(f"{px[i]: <5.1f}", end = " | ")
        for j in range(len(y_val)):
            print(f"{error(py[i], y_val[j][i]):<14.6f}", end = "% | ")
        print()

if __name__ == "__main__":
    main()
```

3.6.d 実行結果:

実行結果を図 16 に示す. また, x の範囲を $4 \leq x \leq 4.5$, y の範囲を $20 \leq y \leq 30$ に拡大した時の実行結果を図 17 に示す.

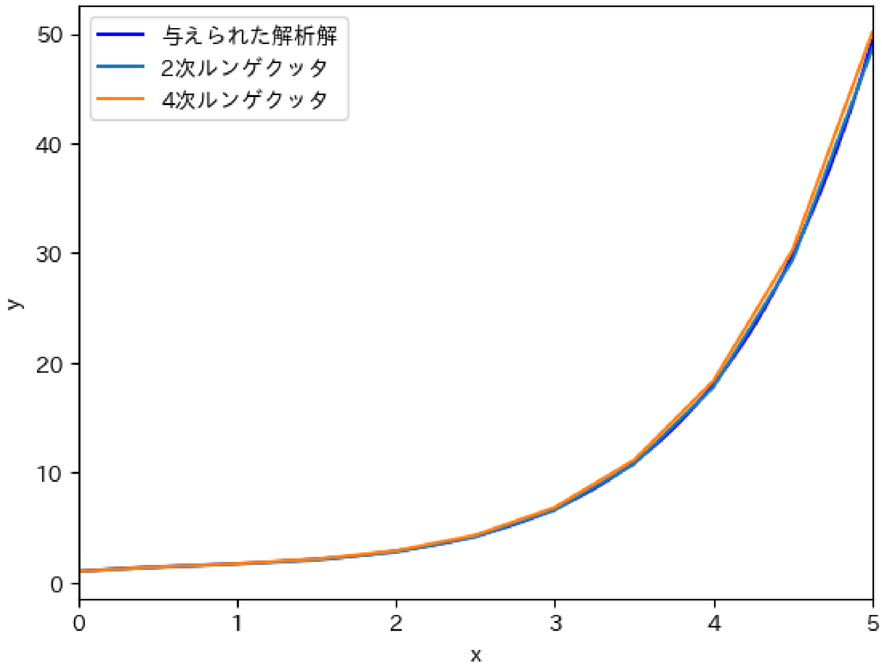

図 16　3.6 実行結果

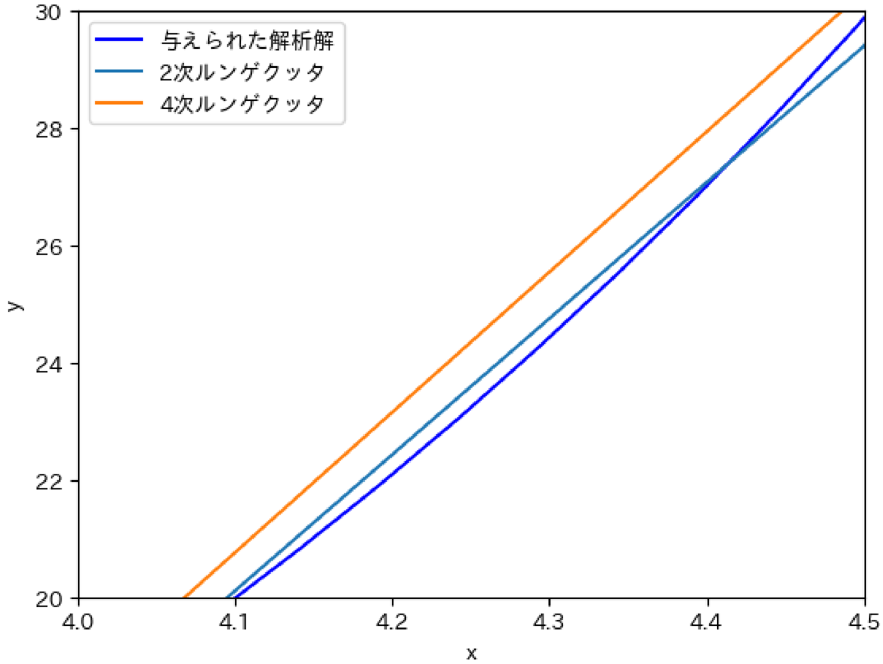

図 17　3.6 実行結果

3.6.e 考察:

図 16, 17 を見るとどちらの曲線も同じように近似していることがわかる.
また, 出力した局所誤差を以下に示す.

解析解と各アルゴリズムとの局所誤差:

x	2次ルンゲクッタ	4次ルンゲクッタ
0.0	0.000000 %	0.000000 %
0.5	1.659293 %	0.008161 %
1.0	3.032242 %	0.034974 %
1.5	3.501690 %	0.369333 %
2.0	3.094476 %	1.042316 %
2.5	2.393752 %	1.564243 %
3.0	1.878608 %	1.768508 %
3.5	1.636493 %	1.751453 %
4.0	1.569872 %	1.650462 %
4.5	1.578392 %	1.546930 %
5.0	1.605335 %	1.470345 %

局所誤差を見ると, 2 次ルンゲクッタ法より 4 次ルンゲクッタ法の方が平均的に精度がよいことがわかる.

課題 3.7

3.7.a 問題:

次の電気回路において, スイッチ S を開いたのち電流の過渡現象は次の微分方程式に従う. ただし, $E = 100[V], R = 15[\Omega], r = 10[\Omega], L = [500mH]$ とする.

$$L\frac{di(t)}{dt} + Ri(t), \text{初期条件 } t = 0 \text{ で } i(0) = \frac{E}{r}[A] \tag{18}$$

これをオイラー法と 4 次のルンゲクッタ法で解き, その結果を以下の解析解とともにグラフで表し比較せよ. ただし, $h = 0.01, 0 \leq t \leq 0.3$ とせよ.

$$i(t) = \frac{E}{R} + (\frac{E}{r} - \frac{E}{R})e^{-\frac{R}{L}t}[A] \tag{19}$$

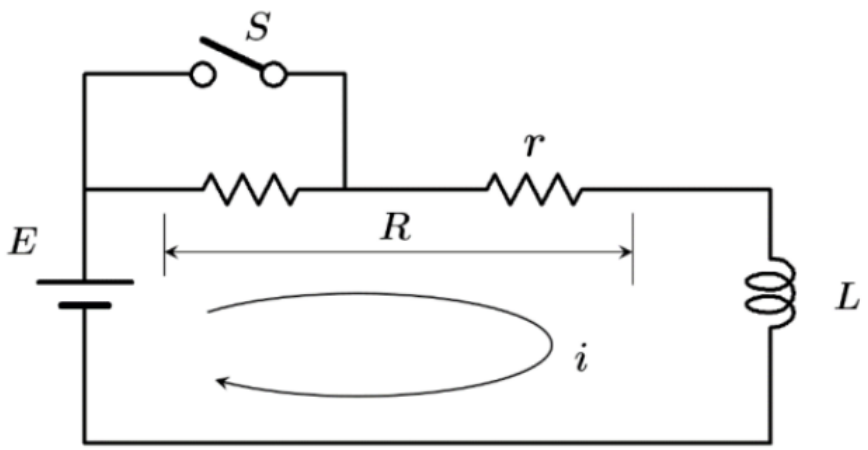

図 18 3.7 電気回路図

3.7.b アルゴリズム:

オイラー法のアルゴリズムは 3.1.b で示した通り, 4 次ルンゲクッタ法のアルゴリズムは 3.4.b, 図 10 に示した通りである.

3.7.c プログラムリスト:

使用したプログラムを以下に示す.

ソースコード 7 演習課題 3.7 Python プログラム

```
#3-7
import numpy as np
```

```python
import math
import sympy as sym
import matplotlib.pyplot as plt
import japanize_matplotlib

#オイラー法
def EulerMethod(t_val, i_val, h, t_max, f, t, i):
    j = 0
    while(True):
        if(t_val[j] >= t_max):
            break
        i_val.append(i_val[j] + h * f.subs([(t, t_val[j]), (i, i_val[j])]))
        t_val.append(t_val[j] + h)
        j += 1

#4次ルンゲクッタ
def RungeKuttaMethod4(t_val, i_val, h, t_max, f, t, i):
    j = 0
    while(True):
        k1 = h * f.subs([(t, t_val[j]), (i, i_val[j])])
        k2 = h * f.subs([(t, t_val[j] + h/2), (i, i_val[j] + k1/2)])
        k3 = h * f.subs([(t, t_val[j] + h/2), (i, i_val[j] + k2/2)])
        k4 = h * f.subs([(t, t_val[j] + h), (i, i_val[j] + k3)])

        if(t_val[j] >= t_max):
            break

        i_val.append(i_val[j] + (k1 + 2*k2 + 2*k3 + k4)/6)
        t_val.append(t_val[j] + h)

        j += 1

def error(est, act): #誤差率を求める関数
    return abs(100 * (est - act) / act)

def main():
    E = 100
    R = 15
    r = 10
    L = 500e-3
    t_val = [[0], [0]]
    i_val = [[E/r], [E/r]]
```

```
    h = 0.01
    t_max = 0.3
    t_min = 0
    t = sym.symbols("t")
    i = sym.symbols("i")
    f = (E - (R * i)) / L
    EulerMethod(t_val[0], i_val[0], h, t_max, f, t, i)
    RungeKuttaMethod4(t_val[1], i_val[1], h, t_max, f, t, i)

    #グラフ
    at=sym.symbols('at')
    expr=E/R + (E/r - E/R) * sym.exp(-R*at/L)
    expr_func=sym.lambdify(at,expr,"numpy")
    px=np.linspace(t_min,t_max, 100)
    py=expr_func(px)

    plt.plot(px,py,color="blue",label=r'解析解')
    plt.plot(t_val[0],i_val[0],label=r'オイラー法')
    plt.plot(t_val[1],i_val[1],"--",label=r'4次ルンゲクッタ')

    plt.xlabel("t")
    plt.ylabel("i")

    plt.xlim(t_min, t_max)

    plt.legend()
    plt.show()

    #局所誤差
    px = np.linspace(t_min, t_max, int(t_max/h+1))
    py = expr_func(px)
    print("解析解と各アルゴリズムとの局所誤差: ")
    print(" t | オイラー法 | 4次ルンゲクッタ |")
    print("------------------------------------------")
    for i in range(len(py)):
        print(f"{px[i]: <5.2f}", end = " | ")
        for j in range(len(i_val)):
            print(f"{error(py[i], i_val[j][i]):<14.6f}", end = "% | ")
        print()

if __name__ == "__main__":
    main()
```

3.7.d 実行結果:

実行結果を図 19 に示す. また, x の範囲を $0.05 \leq x \leq 0.055$, y の範囲を $7.0 \leq y \leq 7.5$ に拡大した時の実行結果を図 20 に示す.

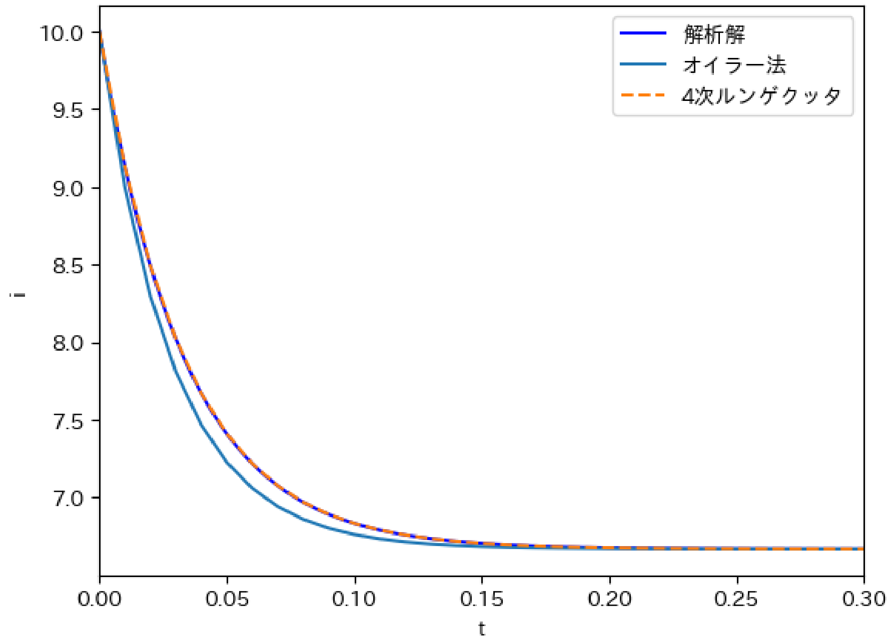

図 19　3.7 実行結果

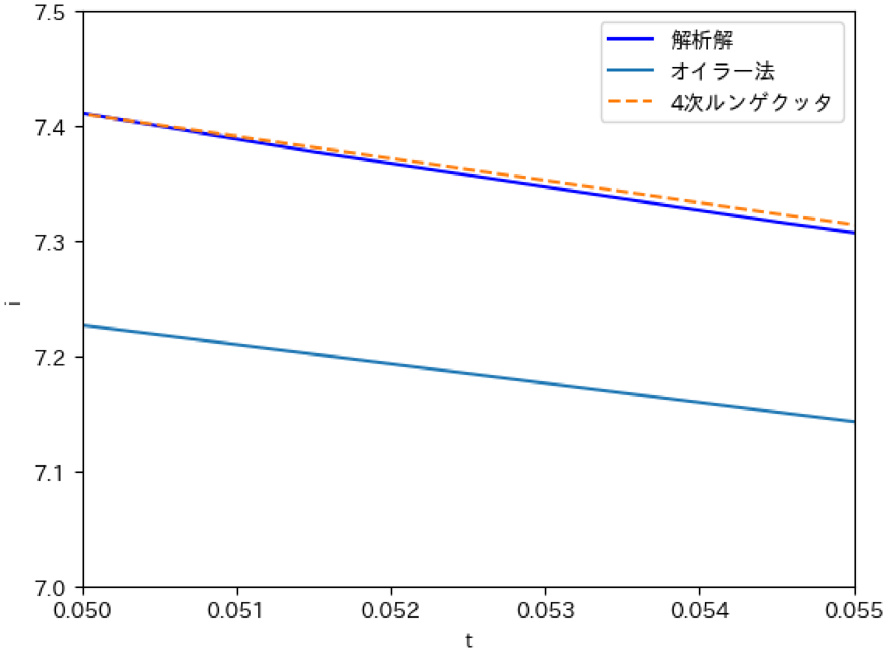

図 20　3.7 実行結果

3.7.e 考察:

図 19, 図 20 を見るとオイラー法も 4 次ルンゲクッタ法もどちらも同じように近似していることがわかる. 出力した局所誤差を以下に示す.

解析解と各アルゴリズムとの局所誤差:

t	オイラー法	4次ルンゲクッタ
0.00	0.000000 %	0.000000 %
0.01	1.489271 %	0.000703 %
0.02	2.307414 %	0.001121 %
0.03	2.641505 %	0.001319 %
0.04	2.654892 %	0.001363 %
0.05	2.476695 %	0.001306 %
0.06	2.200615 %	0.001192 %
0.07	1.889421 %	0.001051 %
0.08	1.581751 %	0.000903 %
0.09	1.298947 %	0.000762 %
0.10	1.050819 %	0.000632 %
0.11	0.840004 %	0.000518 %
0.12	0.665036 %	0.000421 %
0.13	0.522364 %	0.000339 %
0.14	0.407611 %	0.000271 %
0.15	0.316315 %	0.000216 %
0.16	0.244317 %	0.000171 %
0.17	0.187949 %	0.000134 %
0.18	0.144083 %	0.000106 %
0.19	0.110120 %	0.000083 %
0.20	0.083937 %	0.000064 %
0.21	0.063829 %	0.000050 %
0.22	0.048436 %	0.000039 %
0.23	0.036686 %	0.000030 %
0.24	0.027740 %	0.000023 %
0.25	0.020943 %	0.000018 %
0.26	0.015790 %	0.000014 %
0.27	0.011890 %	0.000011 %
0.28	0.008942 %	0.000008 %
0.29	0.006719 %	0.000006 %
0.30	0.005043 %	0.000005 %

これを見ると 4 次ルンゲクッタ法の方がオイラー法と比べて明らかに精度がよいことがわかる. オイラー法の局所誤差を見ると, $(0.01 \leq t \leq 0.10)$ の範囲で誤差が大きくなり, その後誤差は小さくなり安定していることが分かる.
オイラー法の局所誤差が $O(h^2)$, 4 次ルンゲクッタ法の局所誤差が $O(h^5)$ であるため結果は妥当だと言える.

感想

オイラー法や 2, 3, 4 次ルンゲクッタ法, 逆行列法を用いて 1 階微分方程式を解いたが, 各手法の精度の違いを見ることができて良かった.
数値ではなくグラフで結果を表示させることで視覚的に違いを見ることができるので, 場合に応じて数値とグラフを使い分けようと思った.