数値計算-前期期末レポート

HI4 45 番 山口惺司

2024 年 7 月 23 日

課題 2.1

2.1.a 問題:

$\sqrt{7}$の近似値をニュートン法で求めよ.(解：$\sqrt{7} = 2.64575$)

2.1.b アルゴリズム:

ニュートン法のアルゴリズムを図 1 に示す.

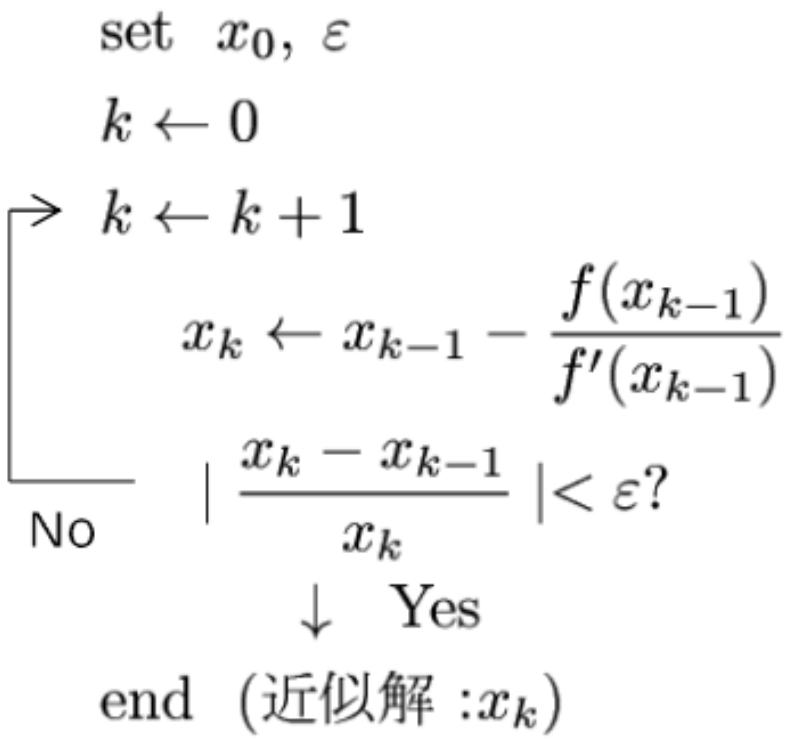

図 1　ニュートン法のアルゴリズム

2.1.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 2.1 Python プログラム

```
1 #2-1
2 import numpy as np
3 import math
```

```
4  import sympy as sym
5  from sympy.plotting import plot
6
7  def NewtonMethod(fa, x, e, a): #ニュートン法による解法
8      k = 0
9      while True:
10         print(k+1,":", x[k])
11         k += 1
12         x.append(float(x[k-1] - fa.subs(a, x[k-1]) / sym.diff(fa, a).subs(a, x[k
               -1])))
13         if abs((x[k] - x[k-1])/x[k]) < e:
14             break;
15
16     print(k+1, ":", x[k])
17     print("2.64574との誤差率:", error(2.64575, x[k]), "%")
18
19 def error(est, act): #誤差率を求める関数
20     return abs(100 * (est - act) / act)
21
22 def main():
23     x = [4]
24     e = 1.0E-5
25     a = sym.symbols("a") #x を変数を表す文字として扱う
26     fa = a ** 2 - 7
27
28     NewtonMethod(fa, x, e, a)
29
30 if __name__ == "__main__":
31     main()
```

2.1.d 実行結果:

$\epsilon = 1.0 \times 10^{-5}, 1.0 \times 10^{-1}$の時の実行結果をそれぞれ図 2, 図 3 に示す.

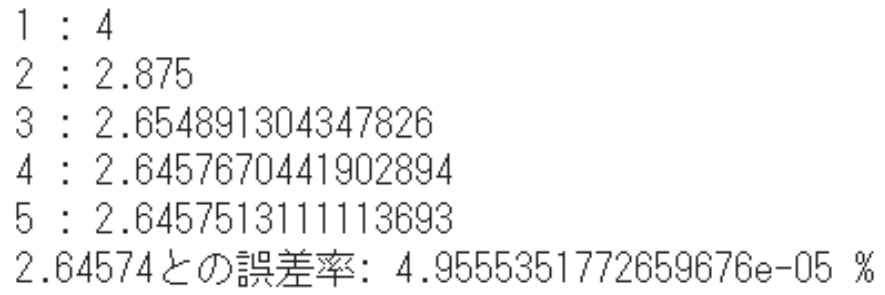

図 2　2.1 実行結果 ($\epsilon = 1.0 \times 10^{-5}$)

図 3　2.1 実行結果 ($\epsilon = 1.0 \times 10^{-1}$)

2.1.e 考察:

$\epsilon = 1.0 \times 10^{-5}, 1.0 \times 10^{-1}$としてニュートン法を用いて$\sqrt{7}$の近似値を求めている.
$\sqrt{7}$は非線形方程式

$$f(x) = x^7 - 7 \tag{1}$$

の解であり,この式をニュートン法の式に代入し,初期値 $x_0 = 4$ を与え,計算すると解析解 $2.64575\cdots$ に収束していく.
実行結果は収束していく様子を示している.
実行結果からϵの値が小さくなるほど反復回数が多くなり,誤差が小さくなることが分かった.

課題 2.2

2.2.a 問題:

$\sqrt{7}$の近似値を 2 分法で求めよ.(解 :$\sqrt{7} = 2.64575$)

2.2.b アルゴリズム:

2 分法のアルゴリズムを図 4 に示す.

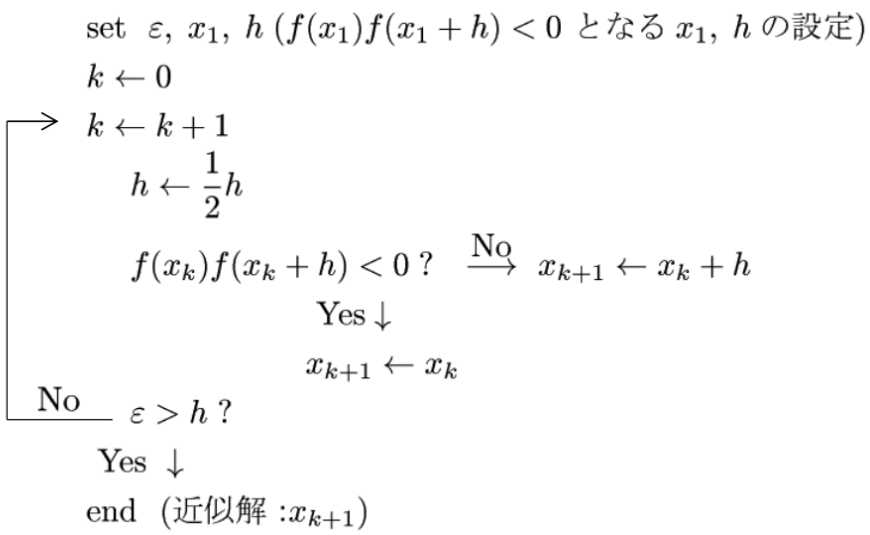

図 4　2 分法のアルゴリズム

2.2.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 2.2 Python プログラム

```
#2-2
import numpy as np
import math
import sympy as sym
from sympy.plotting import plot

def BisectionMethod(fa, x, e, a, h): #二分法による解法
    k = 0
    while True:
        h = 0.5 * h
        if fa.subs(a, x[k]) * fa.subs(a, x[k]+h) < 0:
            x.append(x[k])
        else:
            x.append(x[k] + h)

        if e > h:
            break
```

```
18
19          print(k+1, ":", x[k])
20
21          k += 1
22      print(k+1, ":", x[k])
23      print("2.64574との誤差率:", error(2.64575, x[k]), "%")
24
25  def error(est, act): #誤差率を求める関数
26      return abs(100 * (est - act) / act)
27
28  def main():
29      x = [0]
30      e = 1.0E-5
31      h = 7
32      a = sym.symbols("a") #x を変数を表す文字として扱う
33      fa = a ** 2 - 7
34
35      BisectionMethod(fa, x, e, a, h)
36
37  if __name__ == "__main__":
38      main()
```

2.2.d 実行結果:

初期値 $x_0 = 0$ の時の実行結果を図 5, $x_0 = 8$ の時の実行結果を図 6 に示す.

```
1 : 0
2 : 0
3 : 1.75
4 : 2.625
5 : 2.625
6 : 2.625
7 : 2.625
8 : 2.625
9 : 2.625
10 : 2.638671875
11 : 2.6455078125
12 : 2.6455078125
13 : 2.6455078125
14 : 2.6455078125
15 : 2.6455078125
16 : 2.645721435546875
17 : 2.645721435546875
18 : 2.645721435546875
19 : 2.6457481384277344
20 : 2.6457481384277344
2.64574との誤差率: 7.036090240915545e-05 %
```

図 5　2.2 実行結果 (初期値 $x_0 = 0$)

```
1 : 8
2 : 11.5
3 : 13.25
4 : 14.125
5 : 14.5625
6 : 14.78125
7 : 14.890625
8 : 14.9453125
9 : 14.97265625
10 : 14.986328125
11 : 14.9931640625
12 : 14.99658203125
13 : 14.998291015625
14 : 14.9991455078125
15 : 14.99957275390625
16 : 14.999786376953125
17 : 14.999893188476562
18 : 14.999946594238281
19 : 14.99997329711914
20 : 14.99998664855957
2.64574との誤差率: 82.36165096684223 %
```

図 6　2.2 実行結果 (初期値 $x_0 = 8$)

2.2.e 考察:

$\epsilon = 1 \times 10^{-5}$として 2 分法を用いて$\sqrt{7}$の近似値を求めている.
$\sqrt{7}$は課題 2.1 と同様に非線形方程式

$$f(x) = x^7 - 7 \tag{2}$$

の解であり,この式を 2 分法の式に代入し,初期値 $x_0 = 0, h = 7$ を与え,計算すると解析解 $2.64575\cdots$ に収束していく.
実行結果は収束していく様子を示している.
反復回数は$\epsilon = 1 \times 10^{-5}$の時に 20 回であり,解析解との誤差率は約 $7 \times 10^{-5}\%$であるため,プログラムは正しいと言える.
また,課題 1 の実行結果と比べると,ϵの値が同じとき 2 分法はニュートン法より反復回数と誤差が多くなることが分かる. 図 6 より,2 分法は初期値を適当な値にしないと正しい値が得られないことが分かる.

課題 2.3

2.3.a 問題:

次の積分を区分求積法, 台形公式, シンプソンの公式の各手法を使って計算し, 解析解πと比較し各手法の制度の違いを考察せよ. ただし, 分割数 n は適当に定めよ.

$$S = \int_0^1 \frac{4}{1+x^2} dx \tag{3}$$

2.3.b アルゴリズム:

区分求積法による解法のアルゴリズムを式 (4) と図 7, 台形公式による解法のアルゴリズムを式 (5) と図 8, シンプソンの公式による解法のアルゴリズムを式 (6) と図 9 に示す.

$$S_n = \sum_{k=0}^{n-1} f(x_k)h \tag{4}$$

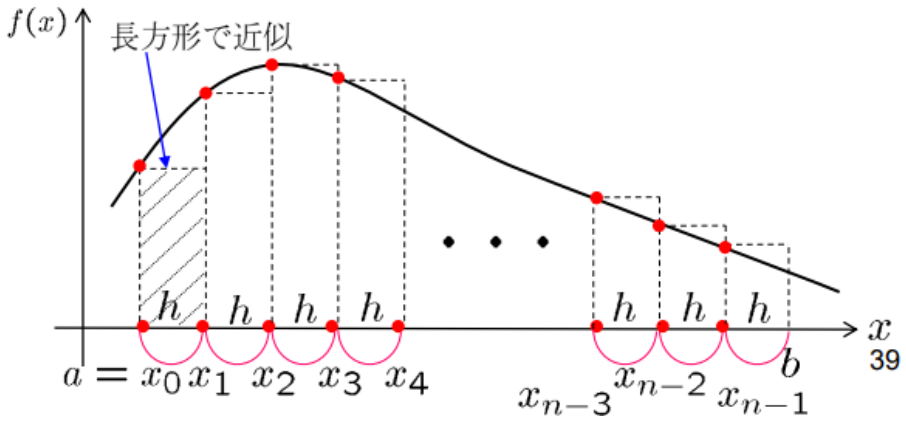

図 7　区分求積法

$$S_n = \sum_{k=0}^{n-1} \frac{h}{2}(f(x_k) + f(x_{k+1})), \text{ただし } x_k = a + kh = a + k\frac{b-a}{n} \tag{5}$$

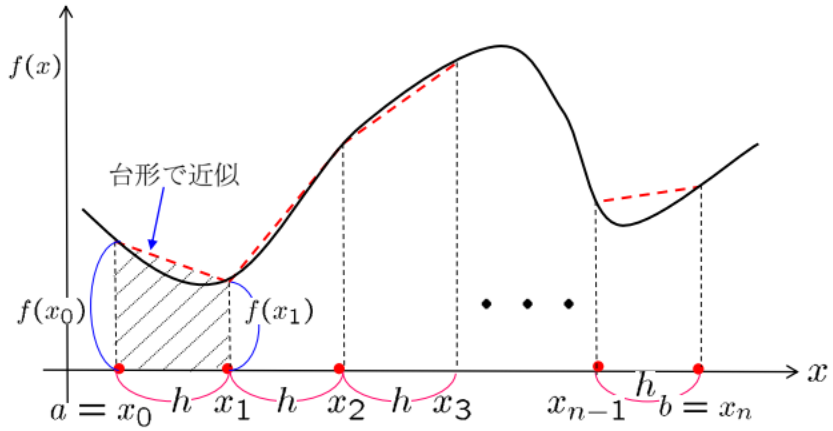

図 8　台形公式

$$S_n = \sum_{k=0}^{\frac{n}{2}-1} \frac{h}{3}(f(x_{2k}) + 4f(x_{2k+1}) + f(x_{2k+2})) \text{ ただし}, x_k = a + kh = a + k\frac{b-a}{n} \tag{6}$$

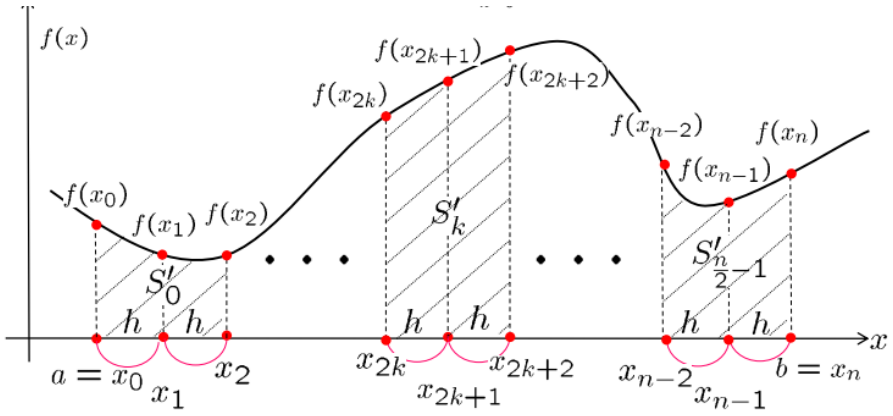

図 9　シンプソンの公式

2.3.c プログラムリスト:

使用したプログラムを以下示す.

ソースコード 3　演習課題 2.3 Python プログラム

```
1 #2-3
2 import numpy as np
3 import math
4 import sympy as sym
5 from sympy.plotting import plot
6
```

```
7  def DefiniteIntegrals(ft, t, a, b, n): #区分求積法による解法
8      h = (b - a)/n
9      xk = []
10     Sn = 0
11
12     for k in range(n):
13         xk.append(a + k * h)
14         Sn += ft.subs(t, xk[k]) * h
15         print(Sn)
16
17     print("区分求積法による解法: ", Sn)
18     print("πとの誤差率: ", error(Sn, math.pi), "%")
19     print()
20
21 def TrapezoidalRule(ft, t, a, b, n): #台形公式による解法
22     h = (b - a)/n
23     xk = []
24     Sn = 0
25
26     for k in range(n+1):
27         xk.append(a + k * h)
28
29     for k in range(n):
30         Sn += (h/2) * (ft.subs(t, xk[k]) + ft.subs(t, xk[k+1]))
31         print(Sn)
32
33     print("台形公式による解法: ", Sn)
34     print("πとの誤差率: ", error(Sn, math.pi), "%")
35     print()
36
37 def SimpsonsRule(ft, t, a, b, n): #シンプソンの公式による解法
38     h = (b - a)/n
39     xk = []
40     Sn = 0
41
42     for k in range(n+1):
43         xk.append(a + k * h)
44
45     for k in range(int(n/2)):
46         Sn += (h/3) * (ft.subs(t, xk[2*k]) + 4*(ft.subs(t, xk[2*k+1])) + ft.subs(
               t, xk[2*k+2]))
47         print(Sn)
```

```
48
49      print("シンプソンの公式による解法: ", Sn)
50      print("πとの誤差率: ", error(Sn, math.pi), "%")
51      print()
52
53  def error(est, act): #誤差率を求める関数
54      return abs(100 * (est - act) / act)
55
56  def main():
57      x = [0]
58      a = 0
59      b = 1
60      n = 5
61      t = sym.symbols("t") #tを変数を表す文字として扱う
62      ft = 4 / (1 + t**2)
63
64      DefiniteIntegrals(ft, t, a, b, n) #区分求積法
65      TrapezoidalRule(ft, t, a, b, n) #台形公式
66      SimpsonsRule(ft, t, a, b, n) #シンプソン
67
68  if __name__ == "__main__":
69      main()
```

2.3.d 実行結果:

分割数 $n = 5, 10, 100$ の時の実行結果をそれぞれ図 10, 図 11, 図 12 に示す.
また, $n = 5, 10$ の時は計算の経過を示す.

```
0.800000000000000
1.56923076923077
2.25888594164456
2.84712123576221
3.33492611381099
区分求積法による解法:  3.33492611381099
πとの誤差率:  6.15399517185277 %

0.784615384615385
1.51405835543767
2.15300358870339
2.69102367478660
3.13492611381099
台形公式による解法:  3.13492611381099
πとの誤差率:  0.212202551823051 %

1.52219274977896
2.69899315875668
シンプソンの公式による解法:  2.69899315875668
πとの誤差率:  14.0883794825332 %
```

図 10　2.3 実行結果 (n=5)

```
0.400000000000000
0.796039603960396
1.18065498857578
1.54762746564000
1.89245505184690
2.21245505184690
2.50657269890572
2.77502907474465
3.01893151376904
3.23992598890716
区分求積法による解法:  3.23992598890716
πとの誤差率:  3.13004727729432 %

0.398019801980198
0.788347296268088
1.16414122710789
1.52004125874345
1.85245505184690
2.15951387537631
2.44080088682518
2.69698029425684
2.92942875133810
3.13992598890716
台形公式による解法:  3.13992598890716
πとの誤差率:  0.0530515845435910 %

0.789591266818990
1.52203555984538
2.16168397093395
2.69896583408026
3.14159261393922
シンプソンの公式による解法:  3.14159261393922
πとの誤差率:  1.26211709486861e-6 %
```

図 11　2.3 実行結果 (n=10)

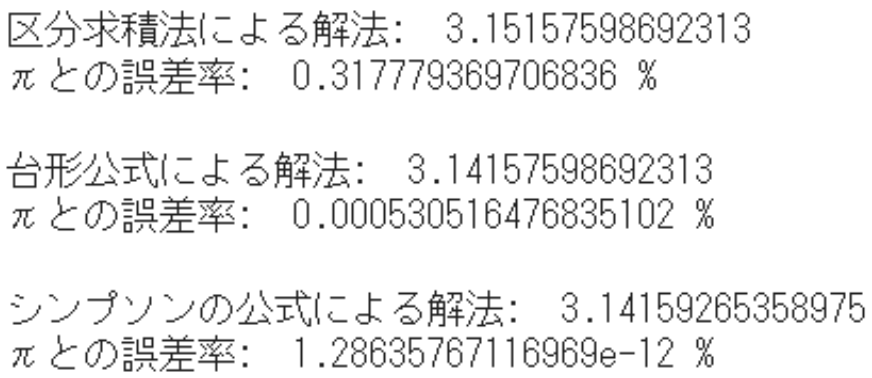

図 12　2.3 実行結果 (n=100)

2.3.e 考察:

実行結果では与えられた式 (3) を, 区分求積法, 台形公式, シンプソンの公式を用い, 分割数 n を変化させ計算したものを表示している.
算出された値を見るとすべてπに近づいて行っていることがわかる.
計算の経過を見ると, $n = 5, 10$ どちらの場合でも区分求積法と台形公式が同じ反復回数でシンプソンの公式がほかの 2 つより反復回数が少ないという結果になった.
これは区分求積法と台形公式の式 (4), (5) から反復回数が $n - 1$ になっている, シンプソンの公式の式 (6) から反復回数が$\frac{n}{2} - 1$ になっているためである.
また, 各手法を用いた時の誤差率は
$n = 5$ の時
区分求積法　　：約 6.2%
台形公式　　：約 0.21%
シンプソンの公式：約 14.1%
$n = 10$ の時
区分求積法　　：約 3.1%
台形公式　　：約 0.053%
シンプソンの公式：約 1.3×10^{-6}%
$n = 100$ の時
区分求積法　　：約 0.32%
台形公式　　：約 0.00053%
シンプソンの公式：約 1.3×10^{-12}%
となり, 分割数 n が大きくなれば誤差も少なくなることが分かる.
また, $n = 5$ の時は, シンプソンの公式 < 区分求積法 < 台形公式という順に, $n = 10, 100$ の時は, 区分求積法 < 台形公式 < シンプソンの公式という順に精度が良くなる.

課題 2.4

2.4.a 問題:

$C-R$ の直列回路に突然直列電圧 E を加えたとき, 加電の瞬時から測って t の時刻の電流 $i(t)$ は次の式で与えられる.

$$i(t) = \frac{E}{R} e^{-\frac{1}{CR}t}[A] \tag{7}$$

電流 i がその最大値 $i_m = \frac{E}{R}$ の $\frac{1}{2}$ になる時間 T をニュートン法で求めよ. ただし, $E = 1[V], R = 100[k\Omega], C = 80[\mu F]$ とする.(解：$T = 5.5452sec$)

2.4.b アルゴリズム:

ニュートン法のアルゴリズムは図 1 に示した通り.

2.4.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 2.4 Python プログラム

```
#2-4
import numpy as np
import math
import sympy as sym
from sympy.plotting import plot

def NewtonMethod(fa, x, e, a): #ニュートン法による解法
    k = 0
    while True:
        print(k+1, ":", x[k])
        k += 1
        x.append(float(x[k-1] - fa.subs(a, x[k-1]) / sym.diff(fa, a).subs(a, x[k
            -1])))
        if abs((x[k] - x[k-1])/x[k]) < e:
            break;

    print("5.5452との誤差率:", error(5.5452, x[k]), "%")


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

```

```
22 def main():
23     x = [1]
24     e = 1.0E-5
25     E = 1
26     R = 100 * 10**3
27     C = 80 * 10**-6
28     t = sym.symbols("t") #x を変数を表す文字として扱う
29     it = (E/R) * math.e ** (-t / (C * R)) - (E/R/2)
30
31     NewtonMethod(it, x, e, t)
32
33 if __name__ == "__main__":
34     main()
```

2.4.d 実行結果:

実行結果を図 13 に示す.

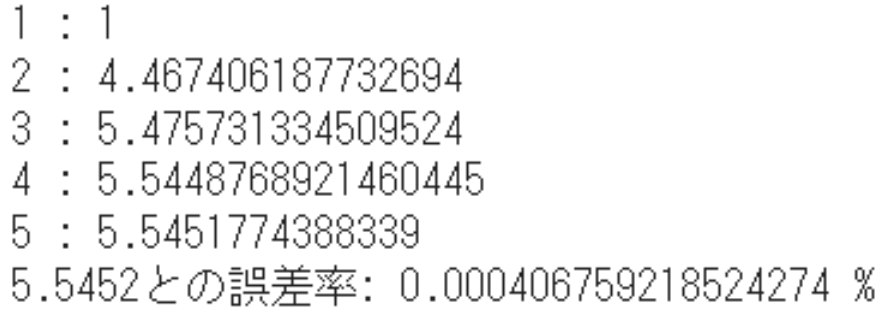

図 13　2.4 実行結果

2.4.e 考察:

問題文より, 電流 i が最大値 $i_m = \frac{E}{R}$ の $\frac{1}{2}$ になればよいので, ニュートン法に代入する式は式のようになる.

$$i(t) = \frac{E}{R}e^{-\frac{1}{CR}t} - \frac{E}{2R}[A] \tag{8}$$

これに初期条件 $E = 1[V], R = 100[k\Omega], C = 80[\mu F], x_0 = 1, \epsilon = 1 \times 10^{-5}$ としてニュートン法で計算する.

算出された値は $5.54517\cdots$ 解析解との誤差率は約 0.0004% となっており, このプログラムは正しいと言える.

また, 実際に式 (8) に初期条件と算出された値を代入してみると式 (9) のようになる.

$$i(5.54517) = \frac{1}{100 \times 10^3}e^{-\frac{5.54517}{80\times10^{-6}\cdot100\times10^3}} - \frac{1}{2 \cdot 100 \times 10^3} \tag{9}$$

これを計算すると,$i(5.54517) = 5.00000465 \times 10^{-6}$となる. これは電流 i の最大値 $i_m = \frac{E}{R} = \frac{1}{100 \times 10^3} = 1 \times 10^{-5}$の$\frac{1}{2}$となっている.
したがってこのプログラムは正しいと言える.

課題 2.5

2.5.a 問題:

$R-L-C$ の直列回路 (図 14) に直流電圧 E を加える.$R^2 = \frac{4L}{C}$の関係があるとき,流れる電流 $i(t)$ は次式で与えられる.(図 15))

$$i(t) = \frac{E}{L} t e^{-\frac{R}{2L}t}[A] \tag{10}$$

電流 i がその最大値 $i_m = \frac{2E}{eR}$の$\frac{1}{2}$になる時間 T_1, T_2を,2 分法で求めよ. ただし,$E = 1[V], R = 1095[\Omega], L = 0.3[H], C = 1[\mu F]$ とする.
(解 : $T_1 = 1.27102 \times 10^{-4} sec, T_2 = 1.46759 \times 10^{-3} sec$)

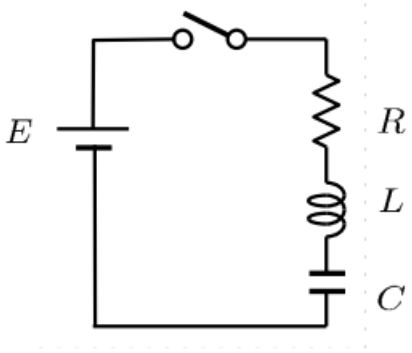

図 14　RLC 回路

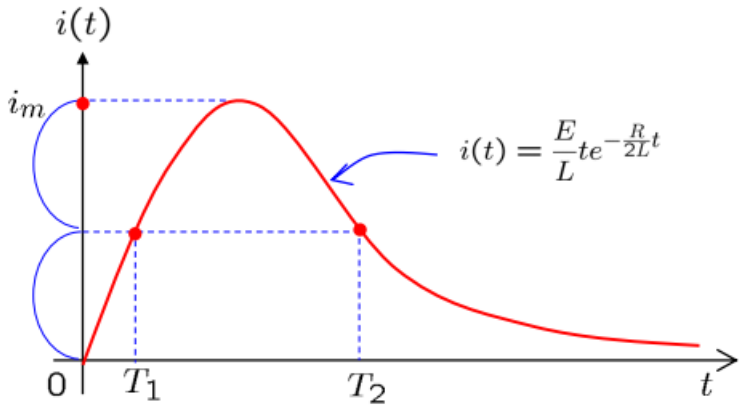

図 15　RLC 回路の過渡応答

2.5.b アルゴリズム:

2 分法のアルゴリズムは図 4 に示した通り.

2.5.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 2.5 Python プログラム

```
1 #2-5
2 import numpy as np
3 import math
```

```
4 import sympy as sym
5 from sympy.plotting import plot
6
7 def BisectionMethod(fa, x, e, a, h): #二分法による解法
8     k = 0
9     while True:
10         h = 0.5 * h
11         if fa.subs(a, x[k]) * fa.subs(a, x[k]+h) < 0:
12             x.append(x[k])
13         else:
14             x.append(x[k] + h)
15
16         print(x[k])
17
18         if e > h:
19             break
20
21         k += 1
22
23 def error(est, act): #誤差率を求める関数
24     return abs(100 * (est - act) / act)
25
26 def main():
27     x1 = [1.0E-4]
28     x2 = [1.0E-3]
29     e = 1.0E-6
30     E = 1
31     R = 1095
32     L = 0.3
33     C = 1 * 10E-6
34     h1 = 1.0E-4
35     h2 = 1.0E-3
36     t = sym.symbols("t") #t を変数を表す文字として扱う
37     it = (E/L) * t * math.e ** ((-t * R) / (2 * L)) - ((2 * E)/(math.e * R)/2)
38
39     print("x の初期値:", x1[0], ", h の初期値:", h1, "の場合")
40     BisectionMethod(it, x1, e, t, h1)
41     print("T1 = 1.27102e-4との誤差率:", round(error(1.27102E-4, x1[-1]),6), "%")
42
43     print("x の初期値:", x2[0], ", h の初期値:", h2, "の場合")
44     BisectionMethod(it, x2, e, t, h2)
45     print("T2 = 1.46759e-3との誤差率:", round(error(1.46759E-3, x2[-1]),6), "%")
```

```
46
47 if __name__ == "__main__":
48     main()
```

2.5.d 実行結果:

実行結果を図 16 に示す.

```
xの初期値: 0.0001 , hの初期値: 0.0001 の場合
0.0001
0.0001
0.000125
0.000125
0.000125
0.000125
0.0001265625
T1 = 1.27102e-4との誤差率: 0.426272 %
xの初期値: 0.001 , hの初期値: 0.001 の場合
0.001
0.001
0.00125
0.001375
0.0014375
0.0014375
0.001453125
0.0014609375
0.00146484375
0.001466796875
T2 = 1.46759e-3との誤差率: 0.054072 %
```

図 16　2.5 実行結果

2.5.e 考察:

実行結果から x, h の初期値を $0.0001, 0.0001$ にした場合は解析解 $T_1 = 1.27102 \times 10^{-4}$に近づき, $0.001, 0.001$ にした場合は解析解 $T_2 = 1.46759 \times 10^{-3}$に近づいていることがわかる.

課題 2.6

2.6.a 問題:

曲面 $z = x^2y^2$ と平面 $x = 0, y = 0, z = 0, x = a, y = b$ で囲まれた体積 V

$$V = \int_0^b \int_0^a x^2y^2 dxdy \tag{11}$$

を区分求積法で計算せよ. ただし,a=10, b=20 として分割数 n, m は適当に定めよ.
(解:V=888889)

2.6.b アルゴリズム:

以下に区分求積法を用いた 2 重積分のアルゴリズムを示す.
長方形領域 $A\{a \leq x \leq b, c \leq y \leq d\}$ の上で定義された 2 変数の連続関数 $f(x, y)$ の 2 重積分 :

$$V = \int \int_A f(x, y)dxdy \tag{12}$$

について考える. まず, x についての積分を 1 次元の積分で用いた区分求積法 (3) 式を適用する.x の積分区間 $[a, b]$ を n 等分すると,

$$V \approx \int_c^d (\sum_{i=0}^{n-1} f(x_i, y)h_x)dy \text{ ただし}, h_x = \frac{b-a}{n} \tag{13}$$

となり, さらに y の積分区間 $[c, d]$ を m 等分して y について区分求積法を適用すると,

$$V = \int_0^b \int_0^a x^2y^2 dxdy \approx \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} f(x_i, y_i)h_xh_y \text{ただし}, h_x = \frac{b-a}{n}, h_y = \frac{d-c}{m} \tag{14}$$

となる. この方法は, 1 次元の積分方法である区分求積法を 2 次元に拡張した方法である. 図 17 にこの方法の説明を示す.
この区分求積法を用いた 2 重積分は, 各省区間 $\{x_i \leq x \leq x_i + h_x, y_j \leq y \leq y_j + h_y\}$ ごとの体積を底面積 h_xh_y, 高さ $f(x_i, y_i)$ の立方体 V_{ij} :

$$V_{ij} = f(x_i, y_j)h_xh_y \tag{15}$$

で近似し, すべての領域においてこの立方体の体積の総和で近似することで, 2 重積分の近似式 (14) 式 :

$$V \approx \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} V_{ij} = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} f(x_i, y_i)h_xh_y \tag{16}$$

が得られる.

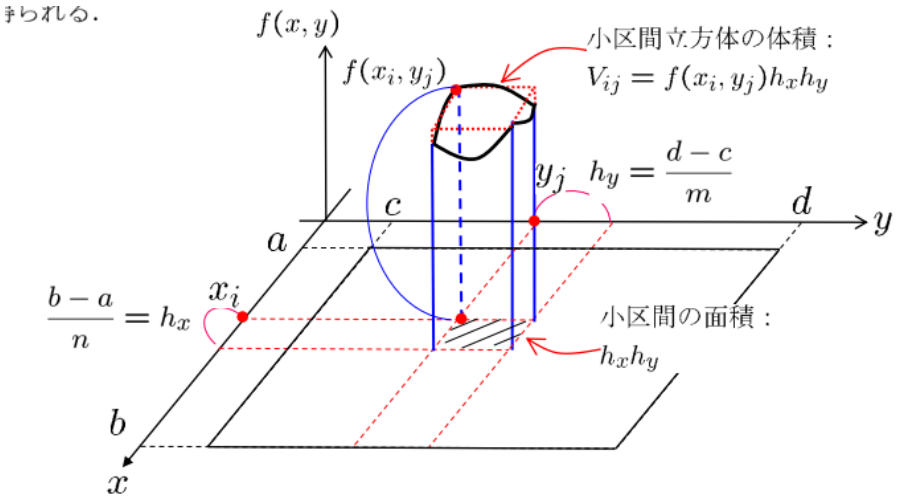

図 17　2 重積分

2.6.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 2.6 Python プログラム

```
1  #2-3
2  import numpy as np
3  import math
4  import sympy as sym
5  from sympy.plotting import plot
6
7  def DefiniteIntegrals(ft, t, a, b, n): #区分求積法による解法
8      h = (b - a)/n
9      xk = []
10     Sn = 0
11
12     for k in range(n):
13         xk.append(a + k * h)
14         Sn += ft.subs(t, xk[k]) * h
15
16     return Sn
17
18 def error(est, act): #誤差率を求める関数
19     return abs(100 * (est - act) / act)
20
21 def main():
22     x = [0]
```

```
23      a = 10
24      b = 20
25      n = 1000
26      m = 1000
27      h = (b - a)/n
28      t = sym.symbols("t") #t を変数を表す文字として扱う
29      u = sym.symbols("u") #t を変数を表す文字として扱う
30      f = t**2 * u**2
31
32      f = DefiniteIntegrals(f, t, 0, a, n) #区分求積法
33      f = DefiniteIntegrals(f, u, 0, b, m) #区分求積法
34      print(f)
35
36      print("V=888889との誤差率:", round(error(888889, f), 5), "%")
37
38  if __name__ == "__main__":
39      main()
```

2.6.d 実行結果:

実行結果を図 18 に示す. また, 分割数 n,m をどちらも 10 にしたときの実行結果を図 19, どちらも 100 にしたときの実行結果を図 20 に示す.

図 18　2.6 実行結果 (n,m = 1000)

図 19　2.6 実行結果 (n,m = 10)

図 20　2.6 実行結果 (n, m = 100)

2.6.e 考察:

区分求積法を 2 回使うことで 2 重積分をしている.
図 18 では解析解 $V = 888889$ との誤差率が約 0.30%なのでプログラムは正しいと言える. 図 18〜20 を見ると分割数 n, m が大きければ大きいほど誤差率が小さくなっていることがわかる.

1　感想

ϵ_0や分割数を変更することで得られる解が変わる. どうやったら正しい解が求められるかを考えて初期値を設定できたためよかった.
課題 2.5 では解が 2 つ出るため, 初期値の設定が大変だった.