数理工学実験
テーマ:関数の補間と数値積分

田中風帆 (1029321151)
実施場所:自宅

実施:2021 年 11 月 8 日
提出:2021 年 11 月 28 日

1　概要

このレポートでは、関数の補間と数値積分についての課題を解き、議論を行う。4.2 節の課題では、問 1 で具体的な関数に対しいくつかの種類の次数の多項式を用いて Lagrange 補間を行い、そのグラフを図示する。問 2 ではいくつかの分割数と中点公式、台形公式、Simpson 公式を組み合わせて具体的な関数に対して定積分を行い、結果をグラフまたは表にまとめる。問 3 では問 2 で求めた積分値と実際の積分値の誤差を求め、グラフまたは表にまとめる。問 4 では問 2,3 の結果に関して議論を行う。4.3 節の課題では、問 1 でいくつかの具体的な関数に対して Gauss 型積分を適用し、結果をグラフにまとめる。問 2 ではある具体的な関数一つに対し、いくつかの積分手法を試して積分値や誤差を求め、その結果の差異を考察する。具体的には、(a) では Gauss 積分および台形公式、Simpson 公式を用いて結果と誤差を求める。(b) では定積分を変数変換したものに対し Gauss 積分を適用し、積分値と誤差を求める。(c) では定積分二つの定積分に分解し、積分値と誤差を求める。(d) では (a),(b),(c) の間で見られる精度の差と、その差が現れる理由について議論する。4.4 節の課題では、有限要素法についての課題を一部解く。

2　4.2 節の課題

1.

ここでは n=2,4,8,16 に対して、与えられた関数 $f(x)$ の $n-1$ 次多項式による Lagrange 補間のグラフを作成する。ただし、関数が定義される区間を (a,b) としたとき、分点 $x_i(i=1...n)$ は $x_i = a + \frac{(b-a)(i-1)}{n-1}$ とする。また、得られたグラフからわかることを述べる。ただし、Lagrange 補間で得られた多項式 $P(x)$ は、x_i よりも細かい分点 y_i をとり、$P(y_i)$ を計算しプロットすることで表現する。

(a)

ここでは $f(x) = \log(x)$ とし、x の範囲を (1,2) とする。結果は以下である。

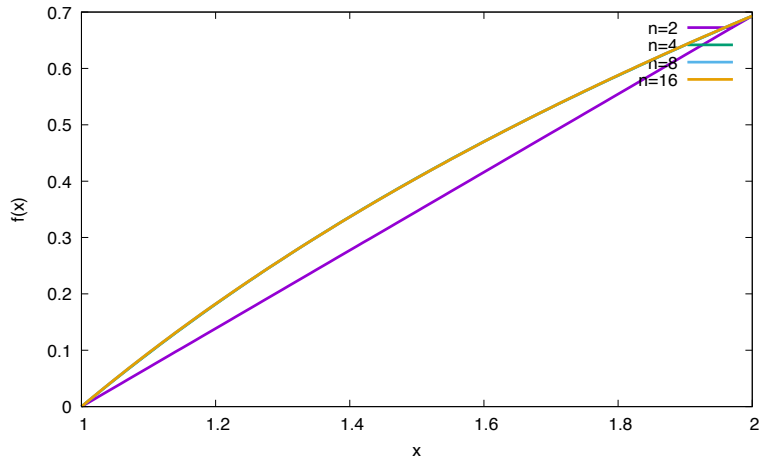

図 1: a の Lagrange 補間. 横軸が x 、縦軸が関数で、区間の分割数によって色分けしている.

以下、解法について詳説する。

Lagrange 補間は、相異なる n 個の点 $x_1 < ... < x_n$ において、関数 f の値 $f(x_i)(i = 1...n)$ が与えられた時、$P(x_i) = f(x_i)$ を満たす $n-1$ 次多項式 $P(x)$ を作り、$x \in [x_1, x_2]$ における $f(x)$ の値を $P(x)$ と推定するというものである。この時、$n-1$ 次の補間多項式 $P(x)$ は一意に定まり、$P(x) = \sum_{i=1}^{n} f(x_i)l_i(x)$(ただし $l_i(x) = \frac{\prod_{j\neq i}(x-x_i)}{\prod_{j\neq i}(x_i-x_j)}$) と表せる。よって、解法は以下のようになる。

まず、$l_i(x) = \frac{\prod_{j\neq i}(x-x_i)}{\prod_{j\neq i}(x_i-x_j)}$ と、Lagrange 補間を行う関数 $f(x)$ を定義する。この小問では $f(x) = \log x$ である。

次の操作を $n = 2, 4, 8, 16$ に対して行う。

積分区間を指定された $n-1$ 個の区間に分割し、分点 $x_i(i = 1...n)$ の値を求める。この小問では、積分区間は (1,2) である。積分区間をさらに細かく離散化する。この時の分点を y_j とする。今回は刻み幅を 0.01 とした。分点 $x_i(i = 1...n)$ の座標を $P(x) = \sum_{i=1}^{n} f(x_i)l_i(x)$ に適用し、分点 y_j における $P(y_j)$ の値を求め、y_j の値とともにファイルに出力する。

以上の操作を行うコードは以下である。

コード 1: (a) の Lagrange 補間を計算するコード

```
1   #include<iostream>
2  #include<vector>
3  #include<cmath>
4  #include<fstream>
5  #include<string>
6  using namespace std;
7
8  //Lagrange 補間において l の値を計算する関数.
9  double l(int n, int i, double x, vector<double> X){
```

```
10      double bunsi=1.0, bunbo=1.0;
11
12      for(int j=0; j<n; j++){
13          if(i == j) continue;
14          bunsi = bunsi * (x−X[j]);
15          bunbo = bunbo * (X[i]−X[j]);
16      }
17
18      return bunsi/bunbo;
19  }
20
21  //Lagrange 補間を行う対象である関数 f を定義.
22  double f(double x){
23      return log(x);
24  }
25
26  int main(){
27      vector<double> X;
28      double dx = 0.01; //刻み幅
29      double a=1.0, b=2.0; //積分区間
30      int step_num = (int)(b−a)/dx;//反復回数
31
32      for(int n=2; n<=16; n=n*2){//それぞれの n において、離散化された
            座標を求める
33          double x = a;
34          X.clear();
35          double x_component = a;
36          for(int k=0; k<n; k++){
37              X.push_back(x_component);
38              x_component += (b−a)/double (n−1);
39          }
40          //出力ファイルを定義
41          ofstream writing_file;
42          string file_name;
43          if(n==2) file_name = "1_a_n2.txt";
44          if(n==4) file_name = "1_a_n4.txt";
45          if(n==8) file_name = "1_a_n8.txt";
46          if(n==16) file_name = "1_a_n16.txt";
47          writing_file.open(file_name, ios::out);
48
49          writing_file << a << "␣" << log(a) << endl;
50
51          //n 区間よりも多い区間に離散化し、積分近似値を求める
52          for(int j=1; j<=step_num; j++){
53              double P = 0;//積分値
54              x += dx;
55              //それぞれの x 座標で積分値を求める
56              for(int i=0; i<n; i++){
57                  P += log(X[i])*l(n, i, x, X);
58              }
59              writing_file << x << "␣" << P << endl;
60          }
61
62          writing_file.close();
63      }
64  }
```

(b)

ここでは $f(x) = \frac{1}{x^3}$ とし、x の範囲を (0.1,2) とする。結果は以下である。使用したコードは (a) の $f(x)$ および積分区間を変更しただけのものであるため省略する。

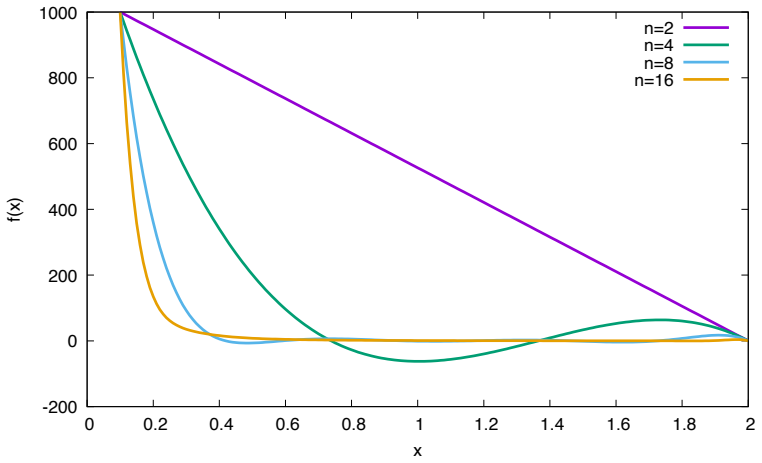

図 2: b の Lagrange 補間. 横軸 x 縦軸が関数で、区間の分割数によって色分けしている.

(c)

ここでは $f(x) = \frac{1}{1+25x^2}$ とし、x の範囲を (-1,1) とする。結果は以下である。使用したコードは (a) の $f(x)$ および積分区間を変更しただけのものであるため省略する。

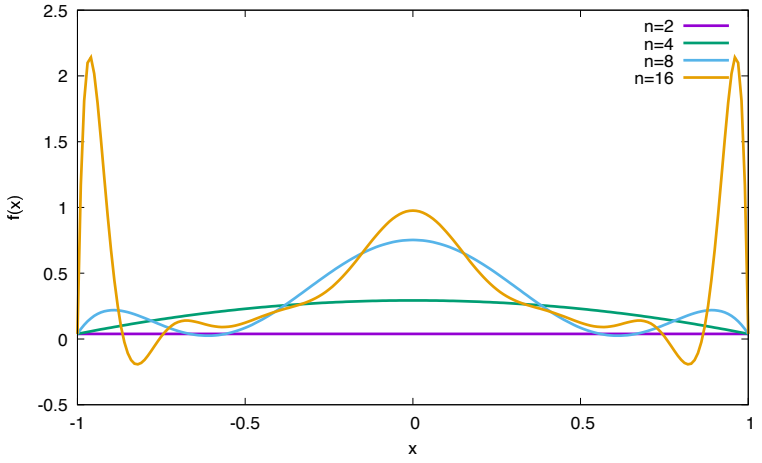

図 3: c の Lagrange 補間. 横軸 x 縦軸が関数で、区間の分割数によって色分けしている.

2.

ここでは、区間 (a,b) における定積分 $I(f) = \int_a^b f(x)dx$ を、分割数 $n = 2^i (i =$

$0, 1, ..., 10)$ の複合中点公式、台形公式、Simpson 公式により求めるコードを作成し、問 (a) から (d) においてそれぞれ与えられる関数 $f(x)$ に対して適用した結果をグラフまたは表にまとめる。(a) ここでは $f(x) = \frac{1}{x}$ とし、x の範囲を (1,2) とする。結果は以下である。

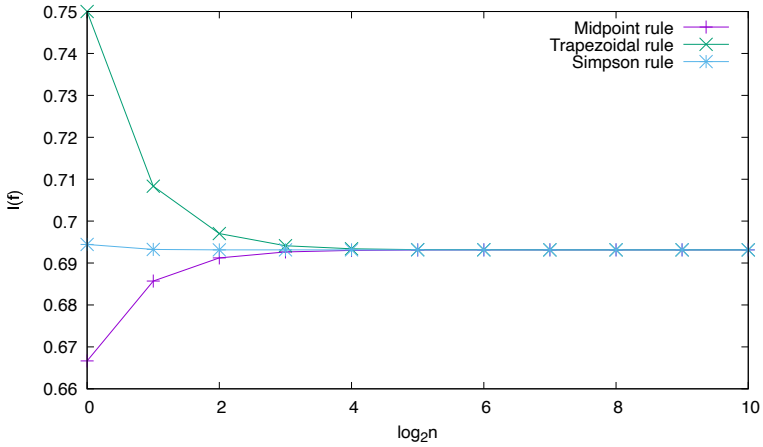

図 4: a の積分近似値. 横軸 i 縦軸積分値で、近似方法によって色分けしている.

手法の詳細を述べる。
積分区間を (a,b)、被積分関数を $f(x)$ とする。
複合中点公式について述べる。$h = \frac{(b-a)}{n}$、$x_i = a + ih(0 \leq i \leq n)$ とおく。積分区間 (a,b) を n 個の小区間 $(x_i, x_{i+1})(0 \leq i \leq n-1)$ に分割し、それぞれの小区間に対して $\int_c^d f(x)dx = (d-c)f(x'_1)$(ただし $x'_1 = \frac{c+d}{2}$) を代入することで、$\int_a^b f(x)dx = h\sum_{i=0}^{n-1} f(\frac{x_i+x_{i+1}}{2})$ を得る。これが複合中点公式である。
複合台形公式について述べる。前例と同じ刻み幅 h と分点 x_i をとり、n 個の小区間 (x_i, x_{i+1}) に台形公式 $\int_c^d f(x)dx = \frac{(d-c)(f(c)+f(d))}{2}$ を適用することで、$\int_a^b f(x)dx = \frac{h}{2}(f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n))$ を得る。これが複合台形公式である。
Simpson 公式について述べる。前例と同じ刻み幅 h と分点 x_i をとり、n 個の小区間 (x_i, x_{i+1}) に Simpson 公式 $\int_c^d f(x)dx = \frac{h'}{3}(f(x'_1)+4f(x'_2)+f(x'_3))$(ただし $x'_i = c+(i-1)\frac{d-c}{2}$)、$h' = \frac{d-c}{2}$) を適用することで、$\int_a^b f(x)dx = \frac{h}{6}(f(x_0)+2\sum_{i=1}^{n-1} f(x_i) + f(x_n) + 4\sum_{i=0}^{n-1} f(\frac{x_i+x_{i+1}}{2}))$ を得る。これが Simpson 公式である。
被積分関数を定義する。この小問では、$f(x) = \frac{1}{x}$ である。積分区間は (1,2) である。以下の操作を、手法 (中点公式、台形公式、Simpson 公式) ごとに、さらに区間の分割数 $n = 2^i(i = 0, ..., 10)$ ごとに行う。

x 座標を n 個の小区間に離散化し、その分点の座標を計算する。この座標を上で述べた中点公式、台形公式、Simpson 公式に代入することで積分値を求める。
以上の操作を行うコードは以下である。

コード 2: (a) の積分を行うコード

```
  #include<iostream>
#include<vector>
#include<cmath>
#include<fstream>
#include<string>
using namespace std;

//被積分関数を定義
double f(double x){
    return 1.0/x;
}

//中点公式を定義
double tyuten(int n, double h, vector<double>x){
    double kinjiti = 0.0;
    for(int i=0; i<n; i++){
        kinjiti += f((x[i]+x[i+1])/2.0);
    }
    return kinjiti * h;
}

//台形公式を定義
double daikei(int n, double h, vector<double>x){
    double kinjiti = 0.0;
    for(int i=1; i<n; i++){
        kinjiti += f(x[i]);
    }
    kinjiti += f(x[0])/2.0;
    kinjiti += f(x[n])/2.0;
    return kinjiti * h;
}

//Simpson 公式を定義
double simpson(int n, double h, vector<double>x){
    double kinjiti = 0.0;
    for(int i=1; i<n; i++){
        kinjiti += f(x[i])*2.0;
    }
    kinjiti += f(x[0]);
    kinjiti += f(x[n]);

    for(int i=0; i<n; i++){
        kinjiti += 4*f((x[i]+x[i+1])/2.0);
    }

    return kinjiti*h / 6.0;
}

//実行
int main(){
    vector<double> x;//離散化した x 座標を格納する
    double a = 1.0, b=2.0;//積分区間

    //出力ファイル
    ofstream tyuten_file, daikei_file, simpson_file;
```

```
56      string filename_tyuten = "1_2_a_tyuten.txt";
57      string filename_daikei = "1_2_a_daikei.txt";
58      string filename_simpson = "1_2_a_simpson.txt";
59
60      tyuten_file.open(filename_tyuten, ios::out);
61      daikei_file.open(filename_daikei, ios::out);
62      simpson_file.open(filename_simpson, ios::out);
63
64      //手法ごとに計算
65      for(int method=0; method<3; method++){
66          //分割数 n ごとに計算
67          for(int i=0; i<=10; i++){
68              int n = pow(2, i);//区間分割数
69              double h = (b−a)/double (n);//刻み幅
70
71              x.clear();
72              double x_i = a;
73              //離散化した x 座標を計算
74              for(int i=0; i<=n; i++){
75                  x.push_back(x_i);
76                  x_i += h;
77              }
78              //計算結果の出力
79              if(method==0){
80                  double ans = tyuten(n, h, x);
81                  tyuten_file << i << "␣" << ans << endl;
82              }
83              if(method==1){
84                  double ans = daikei(n, h, x);
85                  daikei_file << i << "␣" << ans << endl;
86              }
87              if(method==2){
88                  double ans = simpson(n, h, x);
89                  simpson_file << i << "␣" << ans << endl;
90              }
91          }
92      }
93      tyuten_file.close();
94      daikei_file.close();
95      simpson_file.close();
96  }
```

(b) ここでは $f(x) = \exp(5x)$ とし、x の範囲を (-1,1) とする。結果は以下である。コードは、(a) のコードにおいて $f(x)$ の定義と x の範囲のみを変更したものを用いたため、ここでは省略する。

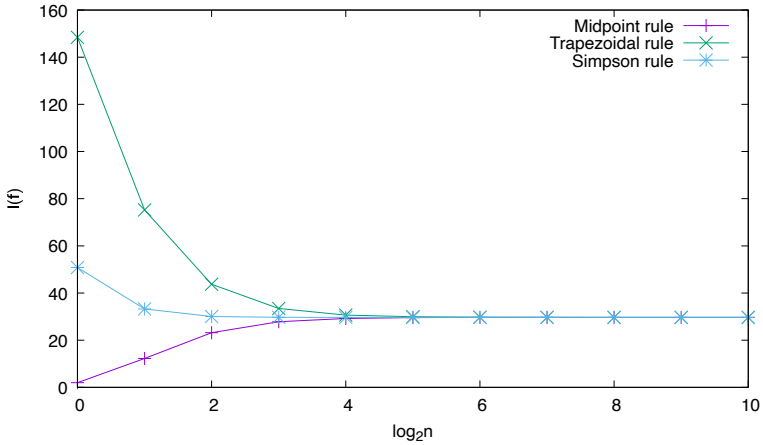

図 5: b の積分近似値. 横軸 i 縦軸積分値で、近似方法によって色分けしている.

(c) ここでは $f(x) = 1 + sin(x)$ とし、x の範囲を $(0,\pi)$ とする。結果は以下である。コードは、(a) のコードにおいて $f(x)$ の定義と x の範囲のみを変更したものを用いたため、ここでは省略する。

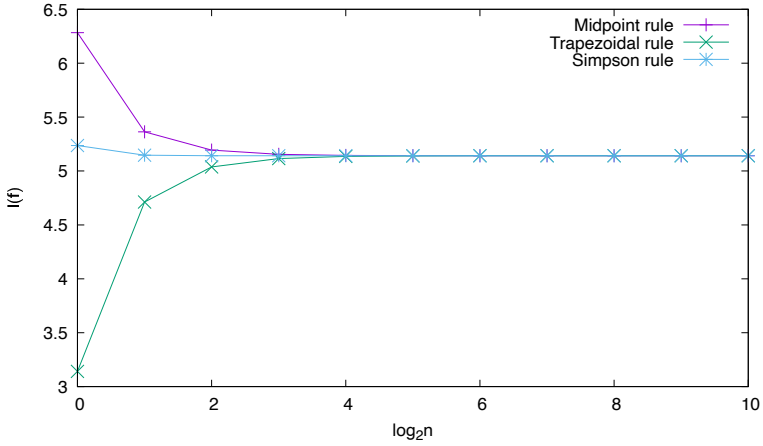

図 6: c の積分近似値. 横軸 i 縦軸積分値で、近似方法によって色分けしている.

(d) ここでは $f(x) = 1 + sin(x)$ とし、x の範囲を $(0,2\pi)$ とする。結果は以下である。コードは、(a) のコードにおいて $f(x)$ の定義と x の範囲のみを変更したものを用いたため、ここでは省略する。

表 1: d の積分近似値. ただし i=j である行の、公式が A である列に位置する数値は、公式 A において区間の分割数を 2^j とした時の積分近似値である.

i の値	中点公式	台形公式	Simpson 公式
$i = 0$	6.28319	6.28319	6.28319
$i = 1$	6.28319	6.28319	6.28319
$i = 2$	6.28319	6.28319	6.28319
$i = 3$	6.28319	6.28319	6.28319
$i = 4$	6.28319	6.28319	6.28319
$i = 5$	6.28319	6.28319	6.28319
$i = 6$	6.28319	6.28319	6.28319
$i = 7$	6.28319	6.28319	6.28319
$i = 8$	6.28319	6.28319	6.28319
$i = 9$	6.28319	6.28319	6.28319
$i = 10$	6.28319	6.28319	6.28319

3.

ここでも、先の 2 番の (a) から (d) までの関数に関して議論を行う。それぞれの関数の定積分 $I(f)$ を分割数 n の数値積分で近似的に求めた値を $I_n(f)$ とし、その誤差を $E_n(f) = I_n(f) - I(f)$ とする。$n = 2^i (i = 0, 1, ..., 10)$ に対する複合中点公式、台形公式、Simpson 公式を使い、(a) から (d) に対する積分の誤差をグラフにまとめる。

(a)

$f(x) = \frac{1}{x}$、x の範囲は (1,2) である。結果は以下である。

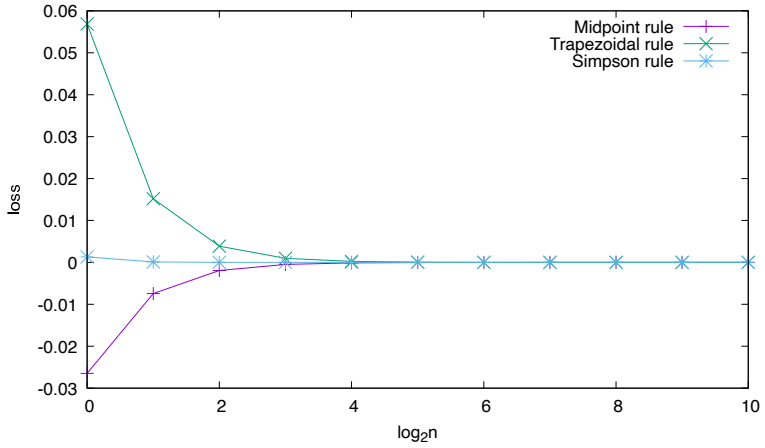

図 7: a の誤差. 横軸 i、 縦軸誤差で、近似方法によって色分けしている.

手法の説明を行う。先の課題 2 と同様の方法で積分の近似値を求め、実際の積分値を引けば誤差が求まる。その値をファイルに出力する。この作業を行うコードは以下である。

コード 3: (a) の積分を行い、誤差を求めるコード

```
  #include<iostream>
#include<vector>
#include<cmath>
#include<fstream>
#include<string>
using namespace std;

//被積分関数を定義
double f(double x){
    return 1.0/x;
}

//中点公式
double tyuten(int n, double h, vector<double>x){
    double kinjiti = 0.0;
    for(int i=0; i<n; i++){
        kinjiti += f((x[i]+x[i+1])/2.0);
    }
    return kinjiti * h;
}

//台形公式
double daikei(int n, double h, vector<double>x){
    double kinjiti = 0.0;
    for(int i=1; i<n; i++){
        kinjiti += f(x[i]);
    }
    kinjiti += f(x[0])/2.0;
    kinjiti += f(x[n])/2.0;
    return kinjiti * h;
}

//Simpson 公式
double simpson(int n, double h, vector<double>x){
    double kinjiti = 0.0;
    for(int i=1; i<n; i++){
        kinjiti += f(x[i])*2.0;
    }
    kinjiti += f(x[0]);
    kinjiti += f(x[n]);

    for(int i=0; i<n; i++){
        kinjiti += 4*f((x[i]+x[i+1])/2.0);
    }

    return kinjiti*h / 6.0;
}

//実行
int main(){
    vector<double> x;//離散化された x 座標を格納
    double a = 1.0, b=2.0;//積分区間
    double real_ans = log(2.0);//実際の積分値

    //出力ファイル
    ofstream tyuten_file, daikei_file, simpson_file;
```

```
57      string filename_tyuten = "1_3_a_tyuten_kai.txt";
58      string filename_daikei = "1_3_a_daikei_kai.txt";
59      string filename_simpson = "1_3_a_simpson_kai.txt";
60
61      tyuten_file.open(filename_tyuten, ios::out);
62      daikei_file.open(filename_daikei, ios::out);
63      simpson_file.open(filename_simpson, ios::out);
64
65      //手法ごとに実行
66      for(int method=0; method<3; method++){
67          //分割数ごとに実行
68          for(int i=0; i<=10; i++){
69              int n = pow(2, i);
70              double h = (b-a)/double (n);
71
72              x.clear();
73              double x_i = a;
74
75              //座標の分点を求める
76              for(int i=0; i<=n; i++){
77                  x.push_back(x_i);
78                  x_i += h;
79              }
80
81              //誤差を計算しファイルに出力
82              if(method==0){
83                  double ans = tyuten(n, h, x)-real_ans;
84                  tyuten_file << i << "␣" << ans << endl;
85              }
86              if(method==1){
87                  double ans = daikei(n, h, x)-real_ans;
88                  daikei_file << i << "␣" << ans << endl;
89              }
90              if(method==2){
91                  double ans = simpson(n, h, x)-real_ans;
92                  simpson_file << i << "␣" << ans << endl;
93              }
94          }
95      }
96      tyuten_file.close();
97      daikei_file.close();
98      simpson_file.close();
99  }
```

(b)

$f(x) = e^{5x}$、x の範囲は (-1,1) である。結果は以下である。コードは (a) のコードにおいて被積分関数の定義と積分範囲を変更するだけであるため、省略する。

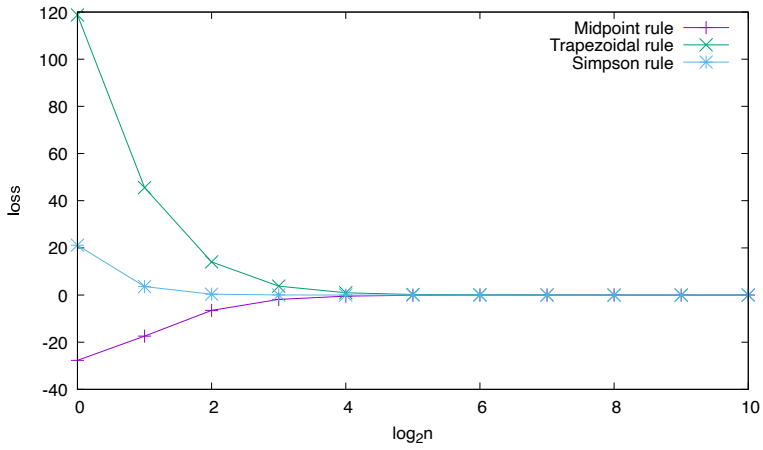

図 8: b の誤差. 横軸 i、 縦軸誤差で、近似方法によって色分けしている.

(c)

$f(x) = 1 + \sin x$、x の範囲は $(0,\pi)$ である。結果は以下である。コードは (a) のコードにおいて被積分関数の定義と積分範囲を変更するだけであるため、省略する。

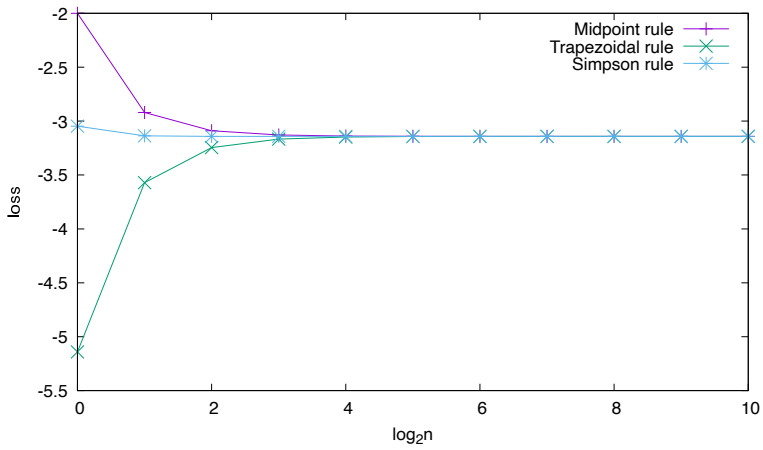

図 9: c の誤差. 横軸 i、 縦軸誤差で、近似方法によって色分けしている.

(d)

$f(x) = 1 + \sin x$、x の範囲は $(0,2\pi)$ である。結果は以下である。コードは (a) のコードにおいて被積分関数の定義と積分範囲を変更するだけであるため、省略する。

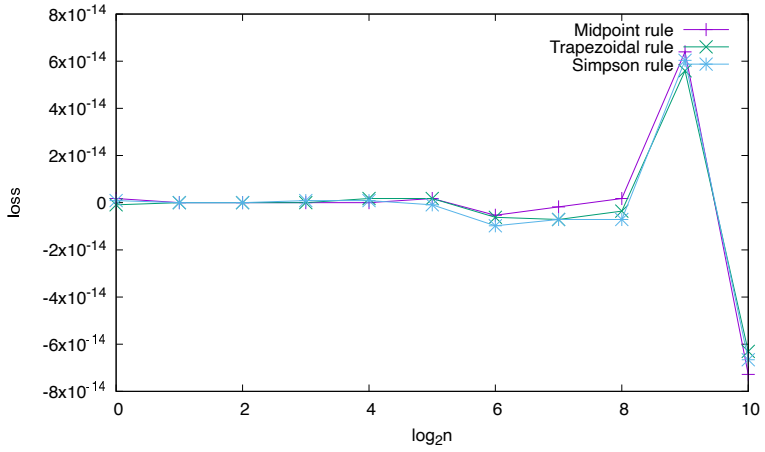

図 10: d の誤差. 横軸 i、 縦軸誤差で、近似方法によって色分けしている.

4.

2 番と 3 番で作成したグラフと表からわかることを述べる。

まず、(d) の積分を除き、区間の分割数が多ければ多いほど積分の誤差は小さくなっていくことがわかる。また、分割数が少ない時の誤差の大きさは Simpson 公式が最も小さく、台形公式が最も大きい。中点公式はその中間である。また、2(d) では公式、分割数にかかわらず積分値は一定である。それにもかかわらず、3(d) では大きな区間分割数を適用した時に誤差が見られる。これらの理由について考察する。

どの公式および分割数を $f(x) = 1 + sin(x)$ に適用しても、その結果は $g(x) = \sin(x)$ にそれらの手法を適用して得られた結果に 2π を加えたものと同じである。$g(x) = \sin(x)$ は $x = \pi$ に関して点対称である。さらに、区間は必ず偶数等分であることから、区間 $(0,\pi)$ と $(\pi,2\pi)$ で誤差を打ち消し合い、正確な値が得られるのだと考えられる。ただし、この場合であっても計算機内部で生じる誤差は消失しないため、これが 3(d) の結果に反映されたのだと考察できる。2(d) で計算機の誤差が反映されていないのは、出力が小数点以下 5 位までであり、誤差が切り捨てられているためであると考えられる。

3　4.3 節の課題

ここでは、Gauss 積分についての問いを解き、結果に対して議論を行う。1 番では 4.2 節の課題 2 で用いた積分と同じものを分割数 $N = 2^i (i = 0, 1, ..., 10)$ の 2 次 Gauss 型積分公式で計算し、結果をグラフにまとめる。2 番では、具体的な定積分 $\int_0^1 \frac{e^{-x}}{\sqrt{x}} dx$ の値を Gauss 型積分によって求める。具体的には、(a) ではそのまま Gauss 型積分を適用し、(b) では $\sqrt{x} = t$ の変数変換を行った

後 Gauss 型積分を適用し、(c) では与式を二つの式に分割した後 Gauss 型積分を適用する。(d) では (a)、(b)、(c) にて得た結果をもとに、手法ごとの精度の違いについて考察を行う。

1.

ここでは、4.2 節の課題 2 で用いた積分と同じもの (被積分関数は、a:$f(x) = \frac{1}{x}$、x の範囲は (1,2)、b:$f(x) = e^{5x}$、x の範囲は (-1,1)、c:$f(x) = 1 + \sin x$、x の範囲は $(0,\pi)$、d:$f(x) = 1 + \sin x$、x の範囲は $(0,2\pi)$) を分割数 $N = 2^i (i = 0, 1, ..., 10)$ の 2 次 Gauss 型積分公式で計算し、結果をグラフまたは表にまとめる。以下が結果である。

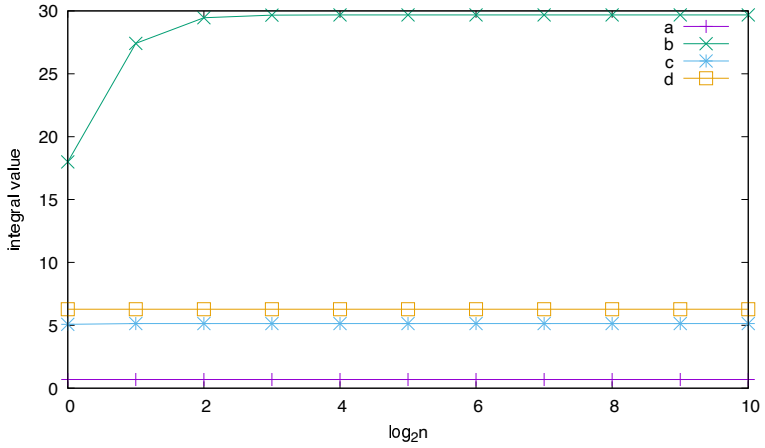

図 11: 2 次の Gauss 型積分で求めた a,b,c,d の積分値. 横軸 i、縦軸積分値で、被積分関数によって色分けしている. ただし、a:$\int_1^2 \frac{1}{x}$、b:$\int_{-1}^1 e^{5x}$、c:$\int_0^\pi 1 + sinx$、d:$\int_0^{2\pi} 1 + sinx$ である。

手法について詳説する。Gauss 型積分公式においても、積分区間を小区間に分割し、それぞれの小区間に低次の積分公式を用いる複合公式を用いる。積分区間が (-1,1) の時、$w_i = \int_{-1}^1 l_i(x)dx$(ただし $l_i(x) = \frac{\prod_{j \neq i}(x - x_i)}{\prod_{j \neq i}(x_i - x_j)}$) を用いて、積分公式 $\int_{-1}^1 f(x)dx = \sum_{i=1}^n w_i f(x_i)$ が成立する。これが n 次の Gauss 積分である。区間 (a,b) における積分を分点 $x_i = a + i(b-a)/n (i = 0, ..., N)$ で N 等分すると、区間 (x_i, x_{i+1}) の積分は変数変換 $y = 2\frac{x - x_i}{x_{i+1} - x_i} - 1$ を用いることで (-1,1) での積分に変換できる。n 次 Gauss 積分の積分点と重みをそれぞれ $y_m, w_m (m = 1, ..., n)$ とすると、$\sum_{i=0}^{N-1} \sum_{m=1}^n f(\frac{(y_m+1)(x_{i+1} - x_i)}{2} + x_i) \frac{x_{i+1} - x_i}{2}$ となる。これが Gauss 型積分公式に対する複合公式である。

この問いを解く方法を述べる。まず、被積分関数 $f(x)$ を定義する。次に、以下の操作を分割数 $n = 2^i (i = 0, ..., 10)$ ごとに行う。

x 座標の分点の座標を計算する。その座標を用い、Gauss 型積分 $\sum_{i=0}^{N-1} \sum_{m=1}^n f(\frac{(y_m+1)(x_{i+1} - x_i)}{2} +$

$x_i)\frac{x_{i+1}-x_i}{2}$ に従って積分値を計算する。
積分 (a) に対してこの作業を行うコードは以下である。(b) から (d) については、被積分関数および積分区間を変更するだけであるため、省略する。

コード 4: (a) の Gauss 積分を行うコード

```
   #include<iostream>
#include<vector>
#include<cmath>
#include<fstream>
#include<string>
using namespace std;

//被積分関数を定義
double f(double x){
    return 1.0/x;
}

//実行
int main(){
    double a = 1.0, b = 2.0;//積分区間
    vector<double> x, w, y;//x,w,y を離散化した分点を格納
    //2 次の Gauss 型積分公式で m=1,2 の時の y_m,w_m の値を格納
    w.push_back(1.0), w.push_back(1.0);
    y.push_back(−1/sqrt(3.0)), y.push_back(1/sqrt(3.0));

    //出力ファイル
    ofstream writing_file;
    string file_name;
    file_name = "2_1_a.txt";
    writing_file.open(file_name, ios::out);

    //分割数を変えて実行
    for(int i=0; i<=10; i++){
        int N = double(pow(2.0, i));//分割数
        double dx = (b − a) / double(N);//小区間の大きさ
        x.clear();
        double x_i = a;

        //分点を格納
        for(int j=0; j<=N; j++){
            x.push_back(x_i);
            x_i += dx;
        }

        //Gauss 型積分を実行
        double ans = 0.0;
        for(int j=0; j<N; j++){
            for(int m=0; m<=1; m++){
                ans += w[m] * f((y[m]+1)*(x[j+1]−x[j])/2.0 + x[j]) * (x[
                    j+1]−x[j]) / 2.0;
            }
        }
        //ファイルに出力
        writing_file << i << "␣" << ans << endl;
    }
    writing_file.close();
}
```

2.
ここでは、積分 $\int_0^1 \frac{e^{-x}}{\sqrt{x}}dx...(*)$ を考える。

(a)
(∗) に対して、積分区間 (0,1) を $N = 2^i (i = 0, 1, ..., 10)$ 等分した小区間に $m = 2, 3$(ただし m は手法の次数) の Gauss 型積分公式を適用し、積分値とその誤差 E_n を求める。ただし、真の積分値として 1.49364826562 を用いる。さらに、Gauss 型積分公式の代わりに台形公式と Simpson 公式を適用し解を求める。
まず、Gauss 型積分を用いた結果が以下である。上の図が積分値を表す図であり、下の図が誤差を表す図である。

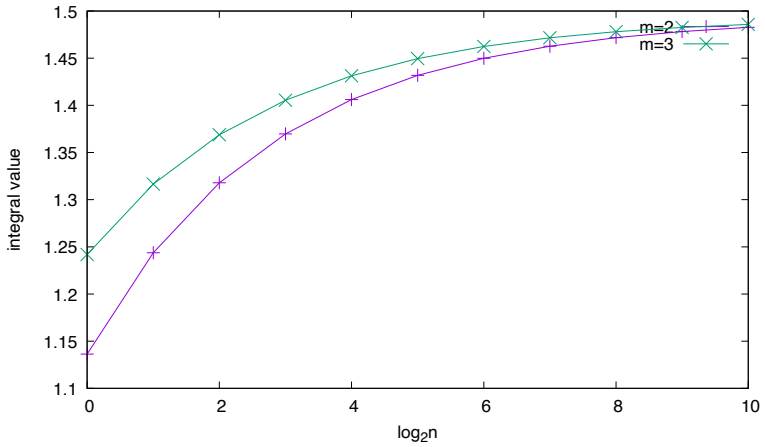

図 12: 縦軸が Gauss 型積分公式を適用して求めた (∗) の値. 横軸が i.Gauss 型積分の次数によって色分けしている.

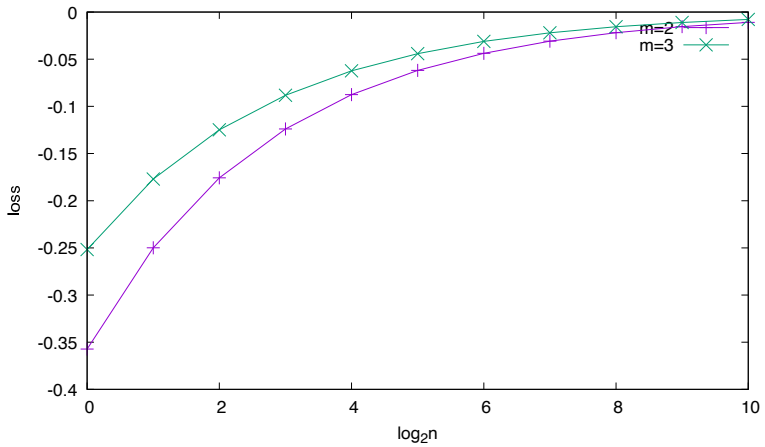

図 13: 縦軸が Gauss 型積分公式を適用して求めた (∗) の値と実際の (∗) の値との誤差. 横軸が i.Gauss 型積分の次数によって色分けしている.

以下が台形公式と Simpson 公式を用いた時の結果である。ただし、$m = 2, 3$ のどちらの場合においても全く同じ結果が得られたため、表は一つのみ掲載する。手法について詳説する。Gauss 型積分は小問 1 で述べた方法、台形公

表 2: $m = 2$,$m = 3$ の時の、(∗) の積分近似値. ただし i=j である行の、公式が A である列に位置する数値は、公式 A において区間の分割数を 2^j とした時の積分近似値である.

i の値	台形公式	Simpson 公式
$i = 0$	inf	inf
$i = 1$	inf	inf
$i = 2$	inf	inf
$i = 3$	inf	inf
$i = 4$	inf	inf
$i = 5$	inf	inf
$i = 6$	inf	inf
$i = 7$	inf	inf
$i = 8$	inf	inf
$i = 9$	inf	inf
$i = 10$	inf	inf

式と Simpson 公式は課題 4.2 で述べた方法とそれぞれ同じである。各分割数 N(Gauss 積分については 2 次、3 次) において、これにより求めた値を近似

された積分値としてファイルに出力し、真の積分値 1.49364826562 を近似された積分値から引いたものを誤差としてファイルに出力する。コードは以下である。

コード 5: Gauss 積分を行い、誤差を求めるコード

```
1  #include<iostream>
2  #include<vector>
3  #include<cmath>
4  #include<fstream>
5  #include<string>
6  #include <iomanip>
7  using namespace std;
8
9  //被積分関数
10 double f(double x){
11     return exp(-x) / sqrt(x);
12 }
13
14 //実行
15 int main(){
16     double a = 0.0, b = 1.0;//積分区間
17     vector<double> x, w, y;//積分に使用する定数
18     ofstream writing_file, writing_file_gosa;
19     string file_name, file_name_gosa;
20
21     for(int M=2; M<=3; M++){//Gauss 積分の次数ごとに計算
22         x.clear(), y.clear(), w.clear();
23         if(M == 2){
24             //w と y の値を格納
25             w.push_back(1.0), w.push_back(1.0);
26             y.push_back(-1/sqrt(3.0)), y.push_back(1/sqrt(3.0));
27             //出力ファイル
28             file_name = "2_2_a_m2.txt";
29             file_name_gosa = "2_2_a_m2_gosa.txt";
30             writing_file.open(file_name, ios::out);
31             writing_file_gosa.open(file_name_gosa, ios::out);
32         }
33         if(M == 3){
34             w.push_back(5.0/9.0),w.push_back(8.0/9.0),w.push_back
                   (5.0/9.0);
35             y.push_back(-sqrt(3.0/5.0)),y.push_back(0.0),y.push_back(
                   sqrt(3.0/5.0));
36             file_name = "2_2_a_m3.txt";
37             file_name_gosa = "2_2_a_m3_gosa.txt";
38             writing_file.open(file_name, ios::out);
39             writing_file_gosa.open(file_name_gosa, ios::out);
40         }
41
42
43         for(int i=0; i<=10; i++){
44             int N = double(pow(2.0, i));//分割数
45             double dx = (b - a) / double(N);//刻み幅
46             x.clear();
47             double x_i = a;
48
49             //x 座標を格納
50             for(int j=0; j<=N; j++){
51                 x.push_back(x_i);
52                 x_i += dx;
53             }
54
```

```
55              //Gauss 積分
56              double ans = 0.0;
57              for(int j=0; j<N; j++){
58                  for(int m=0; m<M; m++){
59                      ans += w[m] * f((y[m]+1)*(x[j+1]−x[j])/2.0 + x[j])
                            * (x[j+1]−x[j]) / 2.0;
60                  }
61              }
62              writing_file << i << "␣" << setprecision(12) <<ans <<
                    endl;
63              //誤差を求める
64              ans −= 1.49364826562;
65              writing_file_gosa << i << "␣" << setprecision(12) <<ans
                    << endl;
66          }
67          writing_file.close();
68          writing_file_gosa.close();
69      }
70  }
```

以下が台形公式および Simpson 公式を用いた場合のコードである。

コード 6: 台形公式および Simpson 公式を用いた積分を行い、誤差を求めるコード

```
1    #include<iostream>
2   #include<vector>
3   #include<cmath>
4   #include<fstream>
5   #include<string>
6   #include <iomanip>
7   using namespace std;
8
9   //被積分関数
10  double f(double x){
11      return exp(−x) / sqrt(x);
12  }
13
14  //台形公式
15  double daikei(int n, double h, vector<double>x){
16      double kinjiti = 0.0;
17      for(int i=1; i<n; i++){
18          kinjiti += f(x[i]);
19      }
20      kinjiti += f(x[0])/2.0;
21      kinjiti += f(x[n])/2.0;
22      return kinjiti * h;
23  }
24
25  //Simpson 公式
26  double simpson(int n, double h, vector<double>x){
27      double kinjiti = 0.0;
28      for(int i=1; i<n; i++){
29          kinjiti += f(x[i])*2.0;
30      }
31      kinjiti += f(x[0]);
32      kinjiti += f(x[n]);
33
34      for(int i=0; i<n; i++){
35          kinjiti += 4*f((x[i]+x[i+1])/2.0);
36      }
37
38      return kinjiti*h / 6.0;
```

```
39 }
40
41 //実行
42 int main(){
43     vector<double> x;//x 座標
44     double a = 0.0, b = 1.0;//積分区間
45     double real_ans = 1.49364826562;//真の積分値
46
47     //出力ファイル
48     ofstream daikei_file, simpson_file, daikei_file_gosa, simpson_file_gosa;
49     string filename_daikei = "2_2_a_daikei.txt";
50     string filename_simpson = "2_2_a_simpson.txt";
51     string filename_daikei_gosa = "2_2_a_daikei_gosa.txt";
52     string filename_simpson_gosa = "2_2_a_simpson_gosa.txt";
53
54     daikei_file.open(filename_daikei, ios::out);
55     simpson_file.open(filename_simpson, ios::out);
56     daikei_file_gosa.open(filename_daikei_gosa, ios::out);
57     simpson_file_gosa.open(filename_simpson_gosa, ios::out);
58
59     for(int method=0; method<2; method++){//手法ごとに繰り返す
60         for(int i=0; i<=10; i++){
61             int n = pow(2, i);//分割数
62             double h = (b−a)/double (n);//刻み幅
63
64             x.clear();
65             double x_i = a;
66             //x 座標を格納
67             for(int i=0; i<=n; i++){
68                 x.push_back(x_i);
69                 x_i += h;
70             }
71
72             if(method==0){
73                 double ans = daikei(n, h, x);//台形公式で計算
74                 double gosa = ans − real_ans;//誤差を求める
75
76                 daikei_file << i << "␣" << setprecision(14) << ans <<
                        endl;
77                 daikei_file_gosa << i << "␣" << setprecision(14) <<
                        gosa << endl;
78             }
79
80             if(method==1){
81                 double ans = simpson(n, h, x);//Simpson 公式で計算
82                 double gosa = ans − real_ans;
83                 simpson_file << i << "␣" << setprecision(14) << ans
                        << endl;
84                 simpson_file_gosa << i << "␣" << setprecision(14) <<
                        gosa << endl;
85             }
86         }
87     }
88     daikei_file.close();
89     simpson_file.close();
90     daikei_file_gosa.close();
91     simpson_file_gosa.close();
92 }
```

(b)

ここでは、(∗) を $\sqrt{x} = t$ で変数変換する。$x = t^2$ であることと $0 < t$ であ

ることから、x が (0,1) の範囲を動く時、積分区間は (0,1) となる。$\frac{dx}{dt} = 2t$ なので、(∗) は $\int_0^1 \frac{e^{-t^2}}{t} 2tdt = \int_0^1 2e^{-t^2} dt$ となる。
次に、上で導いた不定積分に対して、積分区間を $N = 2^i (i = 0, 1, ..., 10)$ 等分した小区間に $m = 2, 3$(ただし m は手法の次数) の Gauss 型積分公式を適用し、積分値とその誤差 E_n を求める。結果のグラフは以下である。

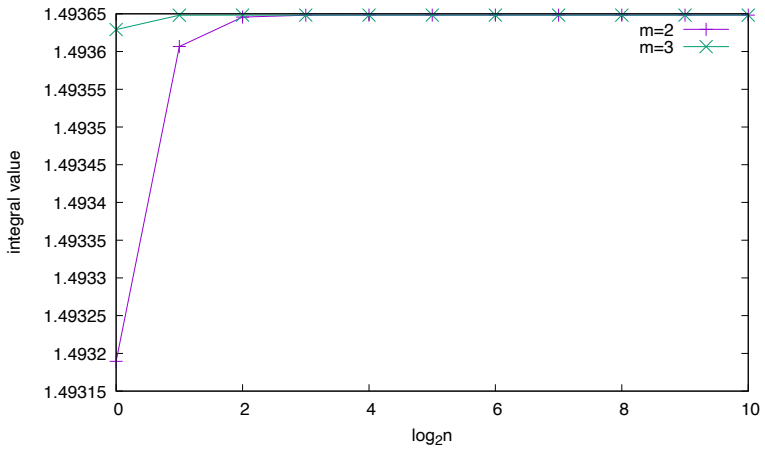

図 14: 縦軸が Gauss 型積分公式を適用して求めた (∗) の値. 横軸が i.Gauss 型積分の次数によって色分けしている.

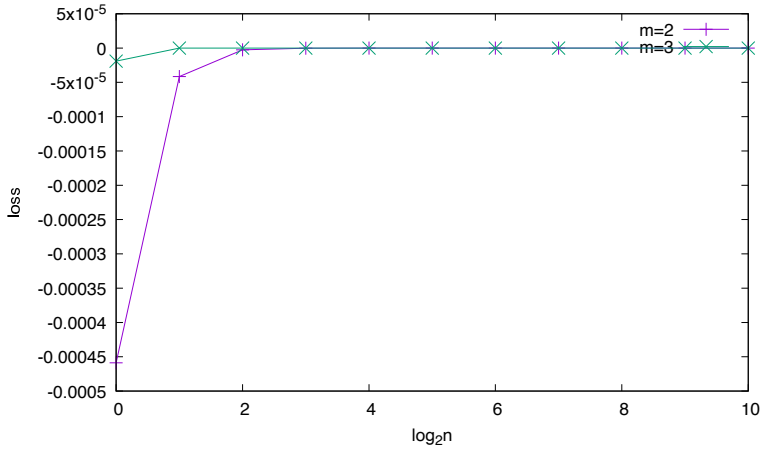

図 15: 縦軸が Gauss 型積分公式を適用して求めた積分値と実際の積分値の誤差. 横軸が i.Gauss 型積分の次数によって色分けしている.

手法について詳説する。被積分関数を $2e^{-t^2}dt$(ただし積分区間は (0,1)) と定義する。他の作業は (a) と同じである。コードは以下である。

コード 7: 変数変換を施した上で積分を行うコード

```
1    #include<iostream>
2  #include<vector>
3  #include<cmath>
4  #include<fstream>
5  #include<string>
6  #include <iomanip>
7  using namespace std;
8
9  //変数変換した被積分関数
10 double f(double x){
11     return 2.0*exp(-x*x);
12 }
13
14 //以降は変数変換しないものと同じ
15 int main(){
16     double a = 0.0, b = 1.0;
17     vector<double> x, w, y;
18     ofstream writing_file, writing_file_gosa;
19     string file_name, file_name_gosa;
20
21     for(int M=2; M<=3; M++){
22         x.clear(), y.clear(), w.clear();
23         if(M == 2){
24             w.push_back(1.0), w.push_back(1.0);
25             y.push_back(-1/sqrt(3.0)), y.push_back(1/sqrt(3.0));
26             file_name = "2_2_b_m2.txt";
27             file_name_gosa = "2_2_b_m2_gosa.txt";
28             writing_file.open(file_name, ios::out);
29             writing_file_gosa.open(file_name_gosa, ios::out);
30         }
31         if(M == 3){
32             w.push_back(5.0/9.0),w.push_back(8.0/9.0),w.push_back
                   (5.0/9.0);
33             y.push_back(-sqrt(3.0/5.0)),y.push_back(0.0),y.push_back(
                   sqrt(3.0/5.0));
34             file_name = "2_2_b_m3.txt";
35             file_name_gosa = "2_2_b_m3_gosa.txt";
36             writing_file.open(file_name, ios::out);
37             writing_file_gosa.open(file_name_gosa, ios::out);
38         }
39
40
41         for(int i=0; i<=10; i++){
42             int N = double(pow(2.0, i));
43             double dx = (b - a) / double(N);
44             x.clear();
45             double x_i = a;
46
47             for(int j=0; j<=N; j++){
48                 x.push_back(x_i);
49                 x_i += dx;
50             }
51
52             double ans = 0.0;
53             for(int j=0; j<N; j++){
54                 for(int m=0; m<M; m++){
55                     ans += w[m] * f((y[m]+1)*(x[j+1]-x[j])/2.0 + x[j])
                           * (x[j+1]-x[j]) / 2.0;
```

```
56                      }
57                  }
58                  writing_file << i << "␣" << setprecision(12) <<ans <<
                        endl;
59                  ans -= 1.49364826562;
60                  writing_file_gosa << i << "␣" << setprecision(12) <<ans
                        << endl;
61              }
62              writing_file.close();
63              writing_file_gosa.close();
64          }
65  }
```

(c)

ここでは、定積分 $(*)$ を $\int_0^1 \frac{e^{-x}}{\sqrt{x}}dx = \int_0^1 \frac{1}{\sqrt{x}} + \int_0^1 \frac{e^{-x}-1}{\sqrt{x}}$ と変形する。第一項の積分を手計算で、第二項の積分を (a)、(b) と同様に離散的に行うことで積分の値を求め、さらにその値と実際の値との誤差 E_n を求める。

$\int_0^1 \frac{1}{\sqrt{x}} = \left[2x^{\frac{1}{2}} + x\right]_0^1 = 2$ である。

$\int_0^1 \frac{e^{-x}-1}{\sqrt{x}}$ を (a),(b) と同様に変数変換なしの場合と $\sqrt{x} = t$ の変数変換ありの場合の両方で離散化して計算し、最後に 2 を足すことで積分値とした。結果のグラフは以下である。

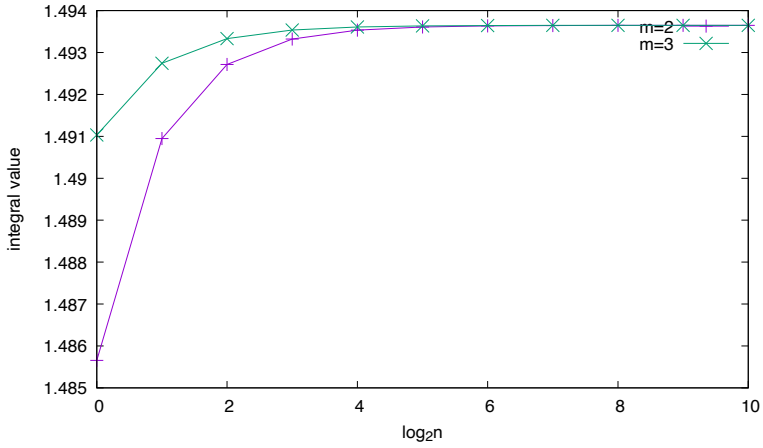

図 16: 縦軸が変数変換なしで求めた $(*)$ の値. 横軸が i.Gauss 型積分の次数によって色分けしている.

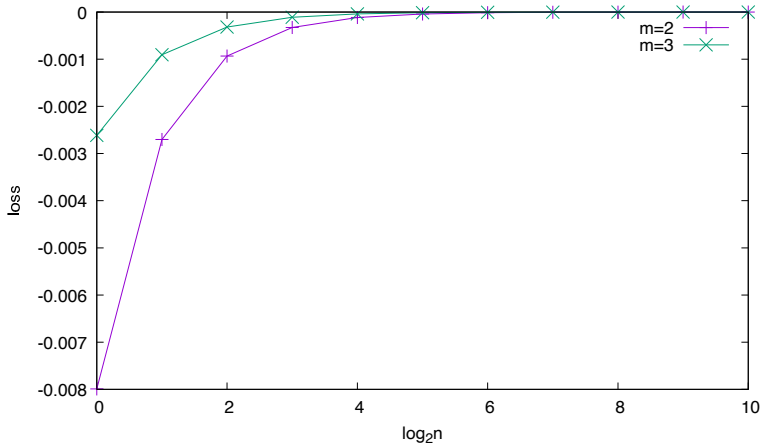

図 17: 縦軸が変数変換なしで求めた (∗) の値と真の値の誤差. 横軸が i.Gauss 型積分の次数によって色分けしている.

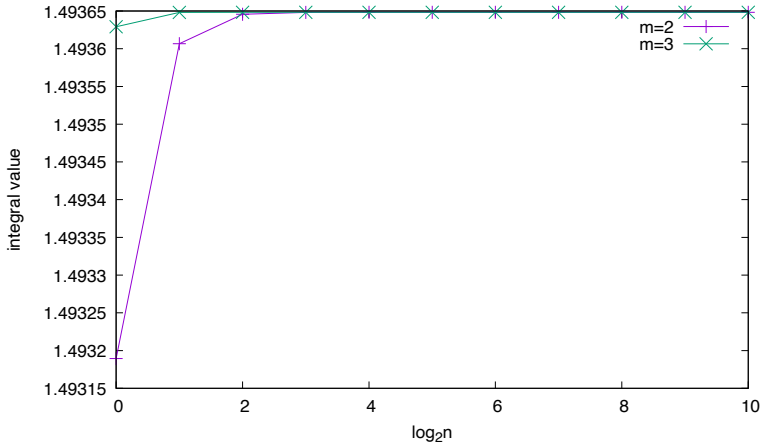

図 18: 縦軸が変数変換ありで求めた (∗) の値. 横軸が i.Gauss 型積分の次数によって色分けしている.

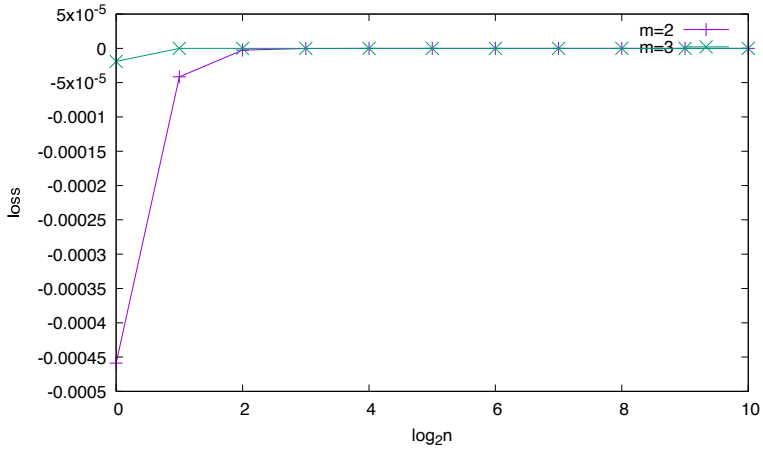

図 19: 縦軸が変数変換ありで求めた (∗) の値と真の値の誤差. 横軸が i.Gauss 型積分の次数によって色分けしている.

手法を詳説する。$\int_0^1 \frac{1}{\sqrt{x}}$ を手計算で求める。この解は 2 である。被積分関数を $\int_0^1 \frac{e^{-x}-1}{\sqrt{x}}$ として、(a)、(b) と同様の方法により Gauss 積分を行う。(b) について、$\int_0^1 \frac{e^{-x}-1}{\sqrt{x}}$ を $t = x^2$ で変数変換すると、積分区間は (0,1) となり、積分は $\int_0^1 2e^{-t^2}dt - 2$ に変換される。Gauss 積分の結果と、先に求めた 2 を足し合わせることで近似された積分値とする。これから真の積分値を引き、誤差とする。近似された積分値および誤差をファイルに出力する。
(a) と同様の計算 (変数変換なし) を行ったコードは以下である。

コード 8: 被積分関数を二つに分割し、変数変換なしで積分を行い誤差を求めるコード

```
1    #include<iostream>
2  #include<vector>
3  #include<cmath>
4  #include<fstream>
5  #include<string>
6  #include <iomanip>
7  using namespace std;
8
9  //変数変換なしの被積分関数
10 double f(double x){
11     return (exp(−x) − 1.0) / sqrt(x);
12 }
13
14 //最後に 2 を足す以外は (a) のコードと同じ
15 int main(){
16     double a = 0.0, b = 1.0;
17     vector<double> x, w, y;
18     ofstream writing_file, writing_file_gosa;
19     string file_name, file_name_gosa;
20
21     for(int M=2; M<=3; M++){
```

```
22          x.clear(), y.clear(), w.clear();
23          if(M == 2){
24              w.push_back(1.0), w.push_back(1.0);
25              y.push_back(−1/sqrt(3.0)), y.push_back(1/sqrt(3.0));
26              file_name = "2_2_c_a_m2.txt";
27              file_name_gosa = "2_2_c_a_m2_gosa.txt";
28              writing_file.open(file_name, ios::out);
29              writing_file_gosa.open(file_name_gosa, ios::out);
30          }
31          if(M == 3){
32              w.push_back(5.0/9.0),w.push_back(8.0/9.0),w.push_back
                    (5.0/9.0);
33              y.push_back(−sqrt(3.0/5.0)),y.push_back(0.0),y.push_back(
                    sqrt(3.0/5.0));
34              file_name = "2_2_c_a_m3.txt";
35              file_name_gosa = "2_2_c_a_m3_gosa.txt";
36              writing_file.open(file_name, ios::out);
37              writing_file_gosa.open(file_name_gosa, ios::out);
38          }
39
40
41          for(int i=0; i<=10; i++){
42              int N = double(pow(2.0, i));
43              double dx = (b − a) / double(N);
44              x.clear();
45              double x_i = a;
46
47              for(int j=0; j<=N; j++){
48                  x.push_back(x_i);
49                  x_i += dx;
50              }
51
52              double ans = 0.0;
53              for(int j=0; j<N; j++){
54                  for(int m=0; m<M; m++){
55                      ans += w[m] * f((y[m]+1)*(x[j+1]−x[j])/2.0 + x[j])
                            * (x[j+1]−x[j]) / 2.0;
56                  }
57              }
58              ans += 2.0;//最後に手計算で求めた 2 を足す
59              writing_file << i << "␣" << setprecision(12) <<ans <<
                    endl;
60              ans −= 1.49364826562;
61              writing_file_gosa << i << "␣" << setprecision(12) <<ans
                    << endl;
62          }
63          writing_file.close();
64          writing_file_gosa.close();
65      }
66  }
```

(b) と同様の計算 (変数変換あり) を行ったコードは以下である。

コード 9: 被積分関数を二つに分割し、変数変換ありで積分を行い誤差を求めるコード

```
1  #include<iostream>
2  #include<vector>
3  #include<cmath>
4  #include<fstream>
5  #include<string>
6  #include <iomanip>
7  using namespace std;
```

```
8
9  //変数変換ありの被積分関数
10 double f(double x){
11     return 2.0 * exp(-x*x) - 2.0;
12 }
13
14 //最後に 2 を足すこと以外は (b) と同じ
15 int main(){
16     double a = 0.0, b = 1.0;
17     vector<double> x, w, y;
18     ofstream writing_file, writing_file_gosa;
19     string file_name, file_name_gosa;
20
21     for(int M=2; M<=3; M++){
22         x.clear(), y.clear(), w.clear();
23         if(M == 2){
24             w.push_back(1.0), w.push_back(1.0);
25             y.push_back(-1/sqrt(3.0)), y.push_back(1/sqrt(3.0));
26             file_name = "2_2_c_b_m2.txt";
27             file_name_gosa = "2_2_c_b_m2_gosa.txt";
28             writing_file.open(file_name, ios::out);
29             writing_file_gosa.open(file_name_gosa, ios::out);
30         }
31         if(M == 3){
32             w.push_back(5.0/9.0),w.push_back(8.0/9.0),w.push_back
                   (5.0/9.0);
33             y.push_back(-sqrt(3.0/5.0)),y.push_back(0.0),y.push_back(
                   sqrt(3.0/5.0));
34             file_name = "2_2_c_b_m3.txt";
35             file_name_gosa = "2_2_c_b_m3_gosa.txt";
36             writing_file.open(file_name, ios::out);
37             writing_file_gosa.open(file_name_gosa, ios::out);
38         }
39
40
41         for(int i=0; i<=10; i++){
42             int N = double(pow(2.0, i));
43             double dx = (b - a) / double(N);
44             x.clear();
45             double x_i = a;
46
47             for(int j=0; j<=N; j++){
48                 x.push_back(x_i);
49                 x_i += dx;
50             }
51
52             double ans = 0.0;
53             for(int j=0; j<N; j++){
54                 for(int m=0; m<M; m++){
55                     ans += w[m] * f((y[m]+1)*(x[j+1]-x[j])/2.0 + x[j])
                           * (x[j+1]-x[j]) / 2.0;
56                 }
57             }
58             ans += 2.0;//手計算で求めた 2 を足す
59             writing_file << i << "␣" << setprecision(12) << ans <<
                   endl;
60             ans -= 1.49364826562;
61             writing_file_gosa << i << "␣" << setprecision(12) <<ans
                   << endl;
62         }
63         writing_file.close();
64         writing_file_gosa.close();
65     }
```

(d)
ここでは、(a),(b),(c) の結果を踏まえて考察を行う。
どの場合においても、大きい n においてはほぼ誤差なく積分値を求められている。2 次の Gauss 積分より 3 次の Gauss 積分の方が小さい n における誤差は小さい。
小さい n における誤差は、(a) が 10^{-2} オーダー、(b) が 10^{-5} オーダーであり、(c) では変数変換しなかったものが 10^{-3} オーダー、変数変換したものが 10^{-5} オーダーとなった。さらに、(a) と (b) の比較および (c) 内での比較により、変数変換したものの方が早く実際の積分値に収束することがわかった。(a) は $x \to 0$ の時被積分関数が無限に発散してしまうため、精度が悪くなる。しかし (b) にて変数変換することによって分母から x が消え、$t \to 0$ で被積分関数が 2 に収束するようになるため、精度が (a) と比べて良くなる。(c) では、変数変換しない場合は分母に x が残ってはいるものの分子が $e^{-x}-1$ となっており、$x \to 0$ で 2 項目の被積分関数 $\frac{e^{-x}-1}{\sqrt{x}}$ が収束する。よって (a) とは違い、n が小さい段階でも誤差は小さくなる。変数変換した場合は (b) と同様の理由でさらに精度が良くなる。

4　4.4 節の課題 (最後の二問以外)

ここでは、有限要素法についての課題を解く。関数 $u(x)$ が n 次元ベクトル空間に属する場合を考える。$u(x)$ を基底関数 $t_i(x)(i=1,...,n)$ と係数 c_i を用いて

$$u(x) = \sum_{i=1}^{n} c_i t_i(x) \tag{1}$$

と書ける。このとき、小問 1 では

区間 (a,b) の分割 $a = x_0 < x_1 < ... < x_{n-1} < x_n = b$ に対して、$h_i = x_i - x_{i-1}(i=1,...,n)$ とし、$t_i(x)$ を

$$t_i(x) = \begin{cases} \frac{x-x_{i-1}}{h_i}(x_{i-1} \le x < x_i) \\ \frac{x_{i+1}-x}{h_{i+1}}(x_i \le x < x_{i+1}) \end{cases} \tag{2}$$

とすると、(1) で表される関数 $u(x)$ は点 $u(x_i) = c_i$ を満たす折れ線関数となることを示す。小問 2 では、関数 $u(x)$ に (1) で表される折れ線関数を代入し、さらに $v = t_i$ を代入して得られる

$$a(u,v) = \int_a^b (p(x)\frac{du}{dx}(x)\frac{dv}{dx}(x) + q(x)u(x)v(x))dx = \int_a^b f(x)v(x)dx \tag{3}$$

が表す n 本の式をまとめて行列表示した $Au = b$ を考える。この解を u_i とし、$\bar{u}(x) = \sum_{j=1}^{n} u_j t_j(x)$ とすると、任意の折れ線関数 $\tilde{v}(x) = \sum_{i=1}^{n} \tilde{v}_i t_i(x)$ に対

して $a(\bar{u},\tilde{v}) = \int_a^b f(x)\tilde{v}(x)dx$ が成り立つことを示す。小問 3 では、Strum-Liouville 型境界値問題

$$-\frac{d}{dx}(e^{-x^2}\frac{du}{dx}(x)) - 6e^{-x^2}u(x) = 0, u(0) = 0, u(1) = -4 \tag{4}$$

を考える。(a) では $u(x) = 8x^3 - 12x$ がこの境界値問題の解であることを確かめる。(b) ではこの境界値問題を、$v(x) = u(x) + 4$ として

$$\begin{cases} -\frac{d}{dx}(p(x)\frac{du}{dx}(x)) + q(x)u(x) = f(x), a < x < b \\ u(a) = u(b) = 0 \end{cases} \tag{5}$$

の形に変形する。(c) では $\frac{dt_i}{dx}(x)$ の値を求める。

1.

ここでは、区間 (a,b) の分割 $a = x_0 < x_1 < ... < x_{n-1} < x_n = b$ に対して、$h_i = x_i - x_{i-1}(i = 1, ..., n)$ とし、$t_i(x)$ を

$$\begin{cases} t_i(x) = \frac{x - x_{i-1}}{h_i}(x_{i-1} \leq x < x_i) \\ \frac{x_{i+1} - x}{h_{i+1}}(x_i \leq x < x_{i+1}) \end{cases} \tag{6}$$

とすると、(1) で表される関数 $u(x)$ は点 $u(x_i) = c_i$ を満たす折れ線関数となることを示す。

このためには、$u(x)$ が $u(x_i) = c_i(i = 0, ..., n)$ を満たし、区間 (x_{i-1}, x_i) において一次関数となることを示せば良い。$t_i(x)$ は $x_{i-1} \leq x < x_{i+1}$ 以外の範囲では 0 となるため、$u(x_i)$ の値を議論するには t_i および t_{i+1} について考察すれば良い。$x = x_i$ において、$t_i(x) = \frac{x_{i+1} - x}{h_{i+1}}$、$t_{i+1}(x) = \frac{x - x_i}{h_{i+1}}$ であるため、$x = x_i$ を代入して $t_i(x_i) = \frac{x_{i+1} - x_i}{x_{i+1} - x_i} = 1$、$t_{i+1}(x_i) = \frac{x_i - x_i}{h_{i+1}} = 0$ となる、よって、$u(x_i) = c_i t_i(x_i) = c_i$ となる。

また、同様の理由により、区間 (x_{i-1}, x_i) における $u(x)$ の形を議論するには $t_i(x)$ および $t_{i-1}(x)$ について考察すれば良い。$x_{i-1} \leq x < x_i$ において、$t_i(x) = \frac{x - x_{i-1}}{h_i}$、$t_{i-1}(x) = \frac{x_i - x}{h_i}$ であるため、$x_{i-1} \leq x < x_i$ における $u(x)$ は、$u(x) = c_i\frac{x - x_{i-1}}{h_i} + c_{i-1}\frac{x_i - x}{h_i} = \frac{1}{h_i}((c_i - c_{i-1})x - c_i x_{i-1} + c_{i-1}x_i)$ となる。これは x に関する一次関数である。

以上より題意は示された。(証明終)

2.

ここでは、関数 $u(x)$ に (1) で表される折れ線関数を代入し、さらに $v = t_i$ を代入して得られる

$$a(u, v) = \int_a^b (p(x)\frac{du}{dx}(x)\frac{dv}{dx}(x) + q(x)u(x)v(x))dx = \int_a^b f(x)v(x)dx \tag{7}$$

に代入して得られる n 本の式をまとめて行列表示した $Au = b$ を考える。この解を u_i とし、$\bar{u}(x) = \sum_{j=1}^n u_j t_j(x)$ とすると、任意の折れ線関数 $\tilde{v}(x) = \sum_{i=1}^n \tilde{v}_i t_i(x)$ に対して $a(\bar{u},\tilde{v}) = \int_a^b f(x)\tilde{v}(x)dx$ が成り立つことを示す。

$a(u,v)$ の定義から、a には線形性がある。$a(\bar{u},t_i) = \int_b^a f(x)t_i(x)dx$ の両辺に $\tilde{v_i}(x)$ をかけて、$\tilde{v_i}(x)a(\bar{u},t_i) = \tilde{v_i}\int_b^a f(x)t_i(x)dx$。これを $i = 1,...,n$ について足し合わせて、$a(\bar{u},\tilde{v}) = \int_b^a f(x)\sum_{i=1}^{i=n}\tilde{v_i}t_i(x)dx = \int_b^a f(x)\tilde{v(x)}dx$ となり、題意は示された。(証明終)

3.

(a)

ここでは、$u(x) = 8x^3 - 12x$ が境界値問題 (4) の解であることを示す。$u(x) = 8x^3 - 12x$ を境界値問題 (4) に代入すると、以下のようになる。

$\frac{du}{dx}(u) = 24x^2 - 12$ より、$\frac{d}{dx}(e^{-x^2}\frac{du}{dx}(x)) = 48xe^{-x^2} - 24xe^{-x^2}(2x^2-1) = 24xe^{-x^2}(-2x^2+3)$

また、$6e^{-x^2}u(x) = 24xe^{-x^2}(2x^2-3)$ となるから、$-\frac{d}{dx}(e^{-x^2}\frac{du}{dx}(x)) - 6e^{-x^2}u(x) = 0$ となる。

さらに、$u(x) = 8x^3 - 12x$ に $x = 0, 1$ を代入し $u(0) = 0$ であること、$u(1) = 4$ であることが確認できる。以上より題意は示された。

(b)

ここでは、境界値問題 (4) を、$v(x) = u(x) + 4$ として

$$\begin{cases} -\frac{d}{dx}(p(x)\frac{du}{dx}(x)) + q(x)u(x) = f(x), a < x < b \\ u(a) = u(b) = 0 \end{cases} \tag{8}$$

の形に変形する。境界値問題 (4) に $v(x) = u(x) + 4$ を代入して、

$$\begin{cases} -\frac{d}{dx}(e^{-x^2}\frac{d(v(x)-4x)}{dx}) - 6e^{-x^2}(v(x) - 4x) = 0 \\ v(0) = 0, v(1) = u(1) + 4 = 0 \\ 0 < x < 1 \end{cases} \tag{9}$$

を得る。これを変形すると、

$$\begin{cases} -\frac{d}{dx}(e^{-x^2}\frac{d(v(x))}{dx}) - 6e^{-x^2}v(x) = -4\frac{d}{dx}e^{-x^2} - 24xe^{-x^2}, 0 < x < 1 \\ v(0) = v(1) = 0 \end{cases} \tag{10}$$

となり、これがこの問の解である。

(c)

t_i の定義 (2) より、$x_{i-1} < x < x_i$ では $\frac{dt_i}{dx} = \frac{d}{dx}\frac{x - x_{i-1}}{h_i} = \frac{1}{h_i} = \frac{1}{x_i - x_{i-1}}$ となり、$x_i < x < x_{i+1}$ では $\frac{dt_i}{dx} = \frac{d}{dx}\frac{x_{i+1} - x}{h_{i+1}} = \frac{-1}{h_{i+1}} = \frac{1}{x_{i+1} - x_i}$ となる。また、$x = x_i, x_{i-1}, x_{i+1}$ では微分不可能で、$x_{i-1} \leq x \leq x_{i+1}$ 以外の x では $t_i(x) = 0$ より $\frac{dt_i}{dx}$ の値も 0 となる。

5 まとめ

このレポートでは、全体を通して関数の補間と数値積分についての課題を解き、議論を行った。4.2 節の課題では、問 1 で具体的な関数に対しいくつか

の種類の次数の多項式を用いて Lagrange 補間を行い、そのグラフを図示した。結果、手法の次数によって補間の結果に差異が出ることがわかった。問 2 ではいくつかの分割数と中点公式、台形公式、Simpson 公式を組み合わせて具体的な関数に対して定積分を行い、結果をグラフまたは表にまとめた。問 3 では問 2 で求めた積分値と実際の積分値の誤差を求め、グラフまたは表にまとめた。問 4 では問 2,3 の結果に関して考察を行った。結果、(d) の積分を除き、区間の分割数が多ければ多いほど積分の誤差は小さくなっていくことがわかった。また、分割数が少ない時の誤差の大きさは Simpson 公式が最も小さく、台形公式が最も大きくなった。中点公式はその中間であった。また、2(d) では公式、分割数にかかわらず積分値は一定であり、それにもかかわらず、大きな区間分割数を適用した時に誤差が見られた。これらはそれぞれ被積分関数の対称性と計算機内部の誤差によるものだと結論づけた。4.3 節の課題では、問 1 でいくつかの具体的な関数に対して Gauss 型積分を適用し、結果をグラフにまとめた結果、分割数の多い方が精度が高いことがわかった。問 2 ではある具体的な関数一つに対し、いくつかの積分手法を試して積分値や誤差を求め、その結果の差異を考察した。具体的には、(a) では Gauss 積分および台形公式、Simpson 公式を用いて結果と誤差を求めた。結果、分割数が多く、Gauss 型積分の次数の大きい方が精度が高くなることがわかった。さらに、台形公式と Simpson 公式では値が発散することがわかった。(b) では定積分を変数変換したものに対し Gauss 積分を適用し、積分値と誤差を求めた。(c) では定積分を二つの定積分に分解し、積分値と誤差を求めた。(d) では (a),(b),(c) の間で見られる精度の差と、その差が現れる理由について考察した。結果、変数変換を施したものは施さなかったものよりも精度が高く、変数変換を施さなかった場合、被積分関数を二つに分割したものは分割しないものに比べて精度が高くなることがわかった。これは $x \to 0$ における極限が発散するか否かに起因すると結論づけた。4.4 節の課題では有限要素法についての課題を解くことで、n 次ベクトル空間に属する関数 $u(x)$ の基底関数が折れ線関数で近似できることを示した。さらに具体的な境界値問題を変形し、有限要素法により解ける形にした。