数値解析レポート No.3

37 番 本間　三暉

1　指数近似

多項式以外の一般的な近似方法の 1 つに指数近似

$$y = a_0 e^{a_1 x} \tag{1}$$

がある．ここでは以下のデータを指数近似してグラフを示す．

(x, y) = (1.5, 8.96), (2.0, 14.78), (2.5, 24.36), (3.0, 40.17)

式 (1) で両辺の対数を取ると，

$$\ln y = \ln a_0 + a_1 x \tag{2}$$

となり，一次近似を行うことができる．線形近似のプログラムを指数近似に対応させたプログラムをソースコード 1 に示す．今回は最小二乗法を利用し，線形近似を実現する．関数 exp_approxmation で配列 a[0],a[1] に格納される値はそれぞれ，式 (2) の a_0, a_1 に対応している．線形近似の一般的な形は，$y = a_0 + a_1 x$ なので，7,18 行目で今回の指数近似の形に項を合わせている．

ソースコード 1　作成した指数近似を行うプログラム [1]

```
1  void exp_approxmation(double *a, double* X, double* Y) {
2
3          int i;
4          double x = 0, y = 0, xx = 0, xy = 0, n = DATANUM;
5
6          for (i = 0; i < n; i++) {
7                  Y[i] = log(Y[i]);
8          }
9          for (i = 0; i < n; i++) {
10                 x += X[i];
11                 y += Y[i];
12                 xx += X[i] * X[i];
13                 xy += X[i] * Y[i];
14         }
15         a[0] = (y * xx - x * xy) / (n * xx - x * x);
16         a[1] = (n * xy - x * y) / (n * xx - x * x);
17
18         a[0] = exp(a[0]);
19         return;
20 }
```

このプログラムにより今回のデータを指数近似した結果，$a_0 \approx 1.99911, a_0 \approx 1.00014$ となった．これらの値を利用し y をグラフ化した様子を図 1 に示す．きれいな指数関数の形になっていることがわかる．

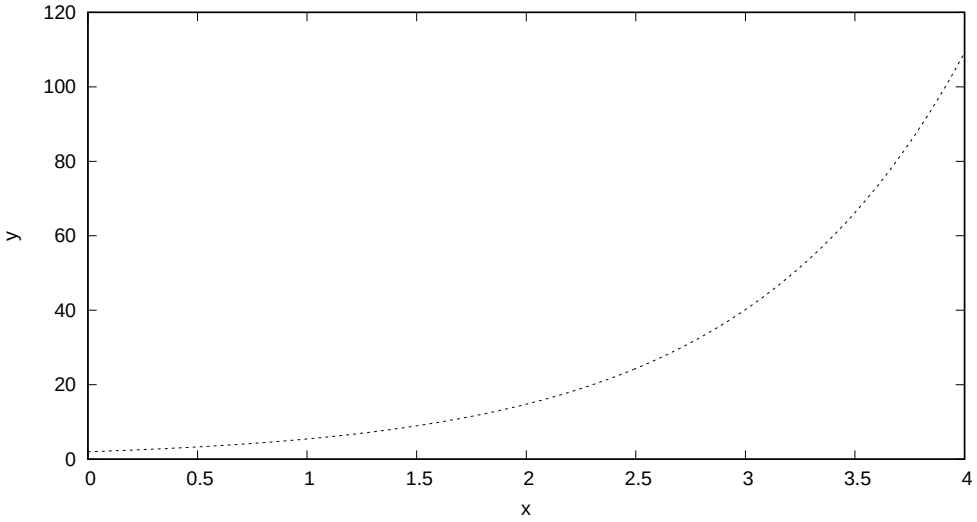

図 1　指数近似により求めた y のグラフ

2　これまでの相互評価課題について

2.1　良いと思うところ，参考になったところ

私は今まで様々な場所で他人のコードを読む機会があったが，読む人のことを考えられてないコードを読んだり，それをフィードバックしたり逆に自分のコードを誰かに評価してもらうことはなかったので学生としてはとても貴重な経験ができたと思う．

また，扱う内容もやる気で満ちている前半にやりごたえのあるものが集まっており，力がついたと思う．

2.2　改善すべきところ，改善案，バグ報告

学生のうちから触れることによって力がつくということで，git について触れてみることを提案したい．しかし，他の人に頼ることが増えてしまう等のデメリットも考えられるので，すり合わせていくべきではある

2.3　疑問・感想

相互評価についての疑問は特にない．感想は 2.1 節と被る部分もあるが，他人のコードを読むことと，他人からのフィードバックを通して自分のプログラミング能力を向上させることができたのでとても良かった．

3　3 種の数値積分の比較

今回は台形公式，シンプソン公式，ロンバーク積分の 3 種の数値積分の方法を (1) 分割数と誤差，(2) 計算量の二つの観点から比較を行う．

これらの三つの手法で定積分を行った結果を比較するプログラムをソースコード 2 に示す．

ソースコード 2　作成した数値積分を行うプログラム

```
1  double trapezoid(double a, double b, int n) {
2
3          double sum = 0, h = (b - a) / n, x = a;
4          int i;
5          for (i = 1; i < n; i++) {
6                  x += h;
7                  sum += f(x);
8          }
9          sum += (f(a) + f(b)) / 2;
10         sum *= h;
11         return sum;
12 }
13
14 double simpson(double a, double b, double n) {
15
16         int i;
17         double sum = 0, h = (b - a) / (2 * n), x = a, mid;
18
19         for (i = 1; i <= n; i++) {
20                 mid = x + h;
21                 sum += (4 * f(mid) + f(x) + f(x + h)) * h / 3;
22                 x += h * 2;
23         }
24         return sum;
25 }
26
27 double romberg(double a, double b, int n) {
28
29         double tmp;
30         double r[n + 1][n + 1];
31         int i, j, k;
32
33         r[1][1] = (b - a) / 2 * (f(a) + f(b));
34
35         for (k = 2; k <= n; k++) {
36                 tmp = 0;
37                 for (i = 1; i <= pow(2, k - 2); i++) {
38                         tmp = tmp + f(a + (2 * i - 1) * (b - a) / pow(2, k - 1));
39                 }
40                 r[k][1] = (r[k - 1][1] + (b - a) / pow(2, k - 2) * tmp) / 2;
41         }
42
43         for (k = 2; k <= n; k++) {
44                 for (j = 2; j <= k; j++) {
45                         r[k][j] = r[k][j - 1] + (r[k][j - 1] - r[k - 1][j - 1]) /
```

```
        (pow(4, j - 1) - 1);
46                  }
47          }
48          return r[n][n];
49  }
```

3.1 分割数と誤差

今回はこのプログラムを使って，$\int_1^3 e^x dx, \int_1^2 x^2 \ln(x+1) dx$ の値を求め，誤差を比較しようと思う．真の値はそれぞれ $\int_1^3 e^x dx \approx 17.367255, \int_1^2 x^2 \ln(x+1) dx \approx 2.2226$ である．図 2,3 は分割数 n と真値との誤差を表したグラフである．これらのグラフを見ると，ロンバーグ法，台形公式，シンプソン公式の順に収束が早いことがわかる．特にロンバーグ法は，分割数が 2 程度でも非常に精度の高い解が得られることがわかる．

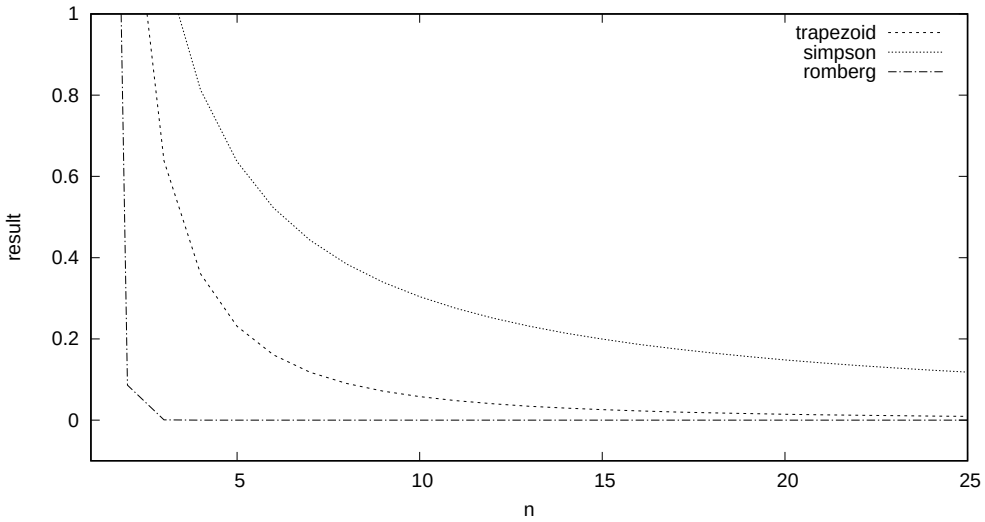

図 2　3 手法で $\int_1^3 e^x dx$ を積分した結果の比較

3.2 計算量

台形公式，シンプソン公式はプログラムも一重ループであることから分かるように，計算量は分割数を n とすると $O(n)$ である．ロンバーク積分の分割数は 2^n となる．ソースコード 2 の 35,37 行目の for 文から，$O(2^n)$ と推測できる．ロンバーグ積分は少ない分割数でも精度の良い解を得ることができるので，分割数はあまり増やさないほうが良い..

参考文献

[1] ウィキペディア-最小二乗法,
”https://ja.wikipedia.org/wiki/最小二乗法”

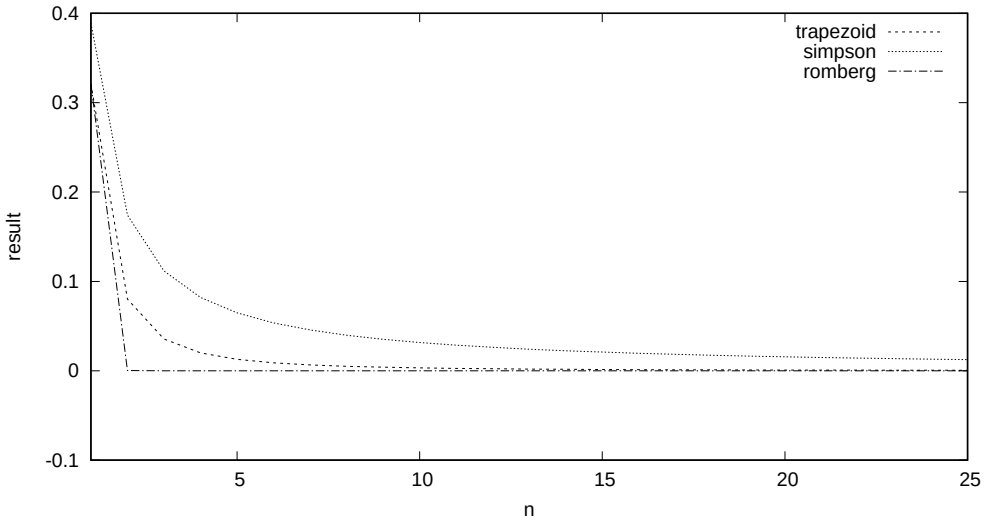

図 3　3 手法で $\int_1^2 x^2 \ln(x+1)dx$ を積分した結果の比較