シミュレーション

4 年電子情報工学科
34 番 横前洸佑

提出日：2019/12/12（木）
提出期限：2019/12/12（木）17:00

1　課題 1

課題 1 では、台形公式、式 (1) を用いて式 (2) について数値積分を行う。さらに、台形公式を使用する際に分割数を 1,2,4,... のように 1/2 ずつ細かくしていき、台形公式で求めた積分値の結果と解析解との関係を報告する。

$$\int_a^b f(x)dx \approx \frac{h}{2}\left[y_0 + 2\sum_{j=1}^{n-1} y_j + y_n\right] \tag{1}$$

$$\int_0^{\frac{\pi}{6}} \frac{dx}{\cos x} \tag{2}$$

1.1　作成したプログラム

今回作成したプログラムをソースコード 1 に示す。

ソースコード 1　課題 1 のプログラム

```
#include <stdio.h>
#include <math.h>

double integration_func(double x);
double trapezoidal_rule(double a, double b, int N);

int main (void){
    int x = 3;
    int N = 1;

    for (; N <= 512; N *= 2){
        double result = trapezoidal_rule(0.0, M_PI / 6, N);
        printf("分割数
            N = %d\n計算結果 = %f\n計算誤差 = %f\n\n", N, result, fabs(0.549306144 - result));
    }
    return 0;
}

//積分される関数
double integration_func (double x){
    double result = 1.0 / cos(x);
    return result;
}

//台形公式
//積分範囲:a -> b
//分割数N
double trapezoidal_rule (double a,double b, int N){
    double h = (b - a) / N;

    double y_0 = integration_func(a);
    double y_n = integration_func(b);
    double tmp = a;
    double y_j = 0.0;
    double res_tmp = 0.0;

    for (int i = 0; i < N-1; i++){
        tmp = tmp + h;
        y_j = integration_func(tmp);
        res_tmp += y_j;
    }

    res_tmp *= 2.0;

    double result = (h / 2.0) * (y_0 + res_tmp + y_n);

    return result;
```

```
}
```

このプログラムでは、分割数 N を 1 から 512 まで計算している。そして、計算結果と式 2 の解析解の $\frac{1}{2}\log_e 3 = 0.549306144$ との差を表示する。なお、積分される関数及び台形公式の計算部分は使いやすくするためにそれぞれ個別の関数にしている。

1.2 プログラムの実行結果

実行結果を以下に示す。

```
分割数 N = 1
計算結果 = 0.564099
計算誤差 = 0.014793

分割数 N = 2
計算結果 = 0.553084
計算誤差 = 0.003778

分割数 N = 4
計算結果 = 0.550256
計算誤差 = 0.000950

分割数 N = 8
計算結果 = 0.549544
計算誤差 = 0.000238

分割数 N = 16
計算結果 = 0.549366
計算誤差 = 0.000059

分割数 N = 32
計算結果 = 0.549321
計算誤差 = 0.000015

分割数 N = 64
計算結果 = 0.549310
計算誤差 = 0.000004

分割数 N = 128
```

```
計算結果 = 0.549307
計算誤差 = 0.000001

分割数 N = 256
計算結果 = 0.549306
計算誤差 = 0.000000

分割数 N = 512
計算結果 = 0.549306
計算誤差 = 0.000000
```

実行結果より、分割数 N が 1/2 になるごとに計算誤差が 1/4 ずつ減っていることがわかる。

1.3 考察

2 課題 2

課題 2 では、シンプソンの公式を用いて式 3 について数値積分を行う。さらに `float` 型と `double` 型で実行し、丸め誤差が現れる刻み幅を調べる。また、刻み幅を 1/2 にした時の誤差の減り方について報告する。

$$\int_0^{\frac{\pi}{2}} \sin x dx \tag{3}$$

2.1 作成したプログラム

今回作成したプログラムをソースコード 2 に示す。

ソースコード 2 課題 2 のプログラム

```c
#include <stdio.h>
#include <math.h>

double integration_func_double(double x);
double simpson_rule_double(double a, double b, int N);

float integration_func_float(float x);
float simpson_rule_float(float a, float b, int N);


int main (void){
    int N = 1;
    for(;N <= 512; N *= 2){
    double result_d = simpson_rule_double(0.0, M_PI / 2, N);
    float result_f = simpson_rule_float(0.0, M_PI / 2, N);
    printf("刻み幅N = %d\nfloat型\n計算結果 = %.10lf\n計算誤差 = %.10lf\ndouble型\n計算結果 = %.10lf\
        n計算誤差 = %.10lf\n\n", N, result_f, fabsf(result_f - 1.0), result_d, fabs(result_d - 1.0));
    }
    return 0;
}

//積分される関数
//double
double integration_func_double (double x){
    double result = sin(x);
```

```
    return result;
}

//シンプソンの公式
//double
double simpson_rule_double (double a,double b, int N){
    double h = (b - a) / N;

    double y0 = integration_func_double(a);
    double Yn = integration_func_double(b);
    //奇数
    double tmp_odd = a - h;
    double Y_odd = 0.0;
    double odd_res_tmp = 0.0;
    //偶数
    double tmp_even = a;
    double Y_even = 0.0;
    double even_res_tmp = 0.0;

    for (int i = 1; i < N; i += 2){
        tmp_odd = tmp_odd + 2 * h;
        Y_odd = integration_func_double(tmp_odd);
        odd_res_tmp += Y_odd;
    }

    for (int i = 2; i < N; i += 2){
        tmp_even = tmp_even + 2 * h;
        Y_even = integration_func_double(tmp_even);
        even_res_tmp += Y_even;
    }

    odd_res_tmp *= 4.0;
    even_res_tmp *= 2.0;

    double result = (h / 3.0) * (y0 + odd_res_tmp  + even_res_tmp + Yn);

    return result;
}

//積分される関数
//float
float integration_func_float (float x){
    float result = sin(x);
    return result;
}

//シンプソンの公式
//float
float simpson_rule_float (float a,float b, int N){
    float h = (b - a) / N;

    float y0 = integration_func_float(a);
    float Yn = integration_func_float(b);
    //奇数
    float tmp_odd = a - h;
    float Y_odd = 0.0;
    float odd_res_tmp = 0.0;
    //偶数
    float tmp_even = a;
    float Y_even = 0.0;
    float even_res_tmp = 0.0;

    for (int i = 1; i < N; i += 2){
        tmp_odd = tmp_odd + 2 * h;
        Y_odd = integration_func_float(tmp_odd);
        odd_res_tmp += Y_odd;
    }

    for (int i = 2; i < N; i += 2){
        tmp_even = tmp_even + 2 * h;
        Y_even = integration_func_float(tmp_even);
        even_res_tmp += Y_even;
    }

    odd_res_tmp *= 4.0;
    even_res_tmp *= 2.0;

    float result = (h / 3.0) * (y0 + odd_res_tmp  + even_res_tmp + Yn);

```

```
    return result;
}
```

このプログラムでは、float 型と double 型でシンプソンの公式を実行する。そして、（計算結果ー真値）の絶対値を表示している。なお、式 3 の真値は 1.0 である。また、刻み幅 N は 1 から 512 まで計算している。

2.2 プログラムの実行結果

実行結果を以下に示す。

```
刻み幅 N = 1
float 型
計算結果 = 0.5235987902
計算誤差 = 0.4764012098
double 型
計算結果 = 0.5235987756
計算誤差 = 0.4764012244

刻み幅 N = 2
float 型
計算結果 = 1.0022798777
計算誤差 = 0.0022798777
double 型
計算結果 = 1.0022798775
計算誤差 = 0.0022798775

刻み幅 N = 4
float 型
計算結果 = 1.0001345873
計算誤差 = 0.0001345873
double 型
計算結果 = 1.0001345850
計算誤差 = 0.0001345850

刻み幅 N = 8
float 型
計算結果 = 1.0000083447
計算誤差 = 0.0000083447
double 型
計算結果 = 1.0000082955
```

```
計算誤差 = 0.0000082955

刻み幅 N = 16
float 型
計算結果 = 1.0000005960
計算誤差 = 0.0000005960
double 型
計算結果 = 1.0000005167
計算誤差 = 0.0000005167

刻み幅 N = 32
float 型
計算結果 = 1.0000001192
計算誤差 = 0.0000001192
double 型
計算結果 = 1.0000000323
計算誤差 = 0.0000000323

刻み幅 N = 64
float 型
計算結果 = 0.9999999404
計算誤差 = 0.0000000596
double 型
計算結果 = 1.0000000020
計算誤差 = 0.0000000020

刻み幅 N = 128
float 型
計算結果 = 1.0000001192
計算誤差 = 0.0000001192
double 型
計算結果 = 1.0000000001
計算誤差 = 0.0000000001

刻み幅 N = 256
float 型
計算結果 = 0.9999997020
計算誤差 = 0.0000002980
```

```
double 型
計算結果 = 1.0000000000
計算誤差 = 0.0000000000

刻み幅 N = 512
float 型
計算結果 = 1.0000002384
計算誤差 = 0.0000002384
double 型
計算結果 = 1.0000000000
計算誤差 = 0.0000000000
```

2.3 考察

3 課題 3,4,5

課題 3 では、オイラー法を用いて式 4 の微分方程式を解く。そして、解析解と数値解を同じグラフにプロットし、オイラー法がどの程度正しいかを報告する。

課題 4 では、課題 3 をホイン法を用いて同様に行う。また、誤差の特徴についても同様に調べる。

課題 5 では、課題 3 をルンゲクッタ法を用いて同様に行う。また、誤差の特徴についても同様に調べる。

$$\frac{du}{dt} = u\,(\text{ただし、}t = 0\text{ のとき }u = 1) \tag{4}$$

3.1 作成したプログラム

今回作成したプログラムをソースコード 3 に示す。

ソースコード 3　課題 3、4、5 のプログラム

```
#include <stdio.h>
#include <math.h>

double diff_equa (double t, double u);
double euler_rule (double t, double u, double h, int step);
double heun_method (double t, double u, double h, int step);
double RK_method (double t, double u, double h, int step);


double calculated_t = 0.0;
double calculated_u = 0.0;
int main (void){
    double t = 0.0;
    double u = 1.0;
    double h = 0.025;
    int step = 40;

    printf("刻み幅 = %f\n", h);

    printf("オイラーの公式\n");
    printf("i, t, u\n");
```

```
    euler_rule(t, u, h, step);

    printf("ホイン法\n");
    printf("i, t, u\n");
    heun_method(t, u, h, step);

    printf("RK法\n");
    printf("i, t, u\n");
    RK_method(t, u, h, step);

    //printf ("t0 = %f, u0 = %f\n", t, u);
   // printf ("t1 = %f, u1 = %f\n", new_t, new_u);

}

//微分方程式
double diff_equa (double t, double u){
    double result = u;
    return result;
}

//オイラーの公式
double euler_rule (double t, double u, double h, int step){
    double old_t = t;
    double old_u = u;
    double new_t = 0;
    double new_u = 0;
    for(int i = 0; i < step; i++){

        new_t = old_t + h;
        new_u = old_u + (h * diff_equa(old_t, old_u));
        old_t = new_t;
        old_u = new_u;
        printf("%d, %f, %f\n", i + 1, new_t, new_u);
    }


    double result = new_u;
    return result;
}

//ホイン法
double heun_method (double t, double u, double h, int step){
    double t_i = t;
    double u_i = u;

    for(int i = 0; i < step; i++){
        double t_i1 = t_i + h;
        double k1 = (h * diff_equa(t_i, u_i));
        double k2 = (h * diff_equa(t_i + h, u_i + k1));

        double u_i1 = u_i + (0.5 * (k1 + k2));
        t_i = t_i1;
        u_i = u_i1;
        printf("%d, %f, %f\n", i + 1, t_i, u_i);
    }
    calculated_t = t_i;
    calculated_u = u_i;

    return 0;
}

//ルンゲ・クッタ法
double RK_method (double t, double u, double h, int step){
    double t_i = t;
    double u_i = u;

    for(int i = 0; i < step; i++){
        double t_i1 = t_i + h;
        double k1 = h * diff_equa(t_i, u_i);
        double k2 = h * diff_equa((t_i + h * 0.5), (u_i + k1 * 0.5));
        double k3 = h * diff_equa((t_i + h * 0.5), (u_i + k2 * 0.5));
        double k4 = h * diff_equa((t_i + h), (u_i + k3));

        double u_i1 = u_i + ((k1 + 2 * k2 + 2 * k3 + k4) / 6);
        t_i = t_i1;
        u_i = u_i1;
        printf("%d, %f, %f\n", i + 1, t_i, u_i);
    }
```

```
    calculated_t = t_i;
    calculated_u = u_i;

    return 0;
}
```

3.2 プログラムの実行結果

実行結果を以下に示す。なお、出力される数値データが多いため、異なる刻み幅のときの同じ t の値のみを示している。

```
刻み幅 = 0.100000
オイラーの公式
i, t, u
1, 0.100000, 1.100000
...
10, 1.000000, 2.593742

ホイン法
i, t, u
1, 0.100000, 1.105000
...
10, 1.000000, 2.714081

RK 法
i, t, u
1, 0.100000, 1.105171
...
10, 1.000000, 2.718280



刻み幅 = 0.050000
オイラーの公式
i, t, u
1, 0.050000, 1.050000
...
20, 1.000000, 2.653298

ホイン法
i, t, u
```

```
1, 0.050000, 1.051250
...
20, 1.000000, 2.717191

RK 法
i, t, u
1, 0.050000, 1.051271
...
20, 1.000000, 2.718282


刻み幅 = 0.025000
オイラーの公式
i, t, u
1, 0.025000, 1.025000
...
40, 1.000000, 2.685064

ホイン法
i, t, u
1, 0.025000, 1.025313
...
40, 1.000000, 2.718004

RK 法
i, t, u
1, 0.025000, 1.025315
...
40, 1.000000, 2.718282
```

3.3 考察