シミュレーション レポート

第1回～第5回

提出期限　2020年11月19日

組番号　408
学籍番号　17406

氏名　金澤雄大

1　目的

シミュレーションの授業の理解度を測るために, 次の 5 つのアルゴリズムについてプログラムを作成することを目的とする. また作成したプログラムの誤差や収束の様子を比較し, 考察することを目的とする.

1. 台形公式
2. シンプソンの公式
3. オイラー法
4. ホイン法

2　実験環境

実験環境を表 1 に示す.gcc は WIndows 上の WSL(Windows Subsystem for Linux) で動作するものを用いる.

表 1: 実験環境

CPU	AMD Ryzen 5 3600 6-Core Processor
メモリ	16.0GB DDR4
OS	Microsoft Windows 10 Home
gcc	(Ubuntu 9.3.0-17ubuntu1〜20.04) 9.3.0
Make	GNU Make 4.2.1

3　課題 1

本章では課題 1 における課題内容, プログラムの説明, 実験結果, 考察の 4 つについて述べる.

3.1　課題内容

課題 1 の課題内容は台形公式を用いて式 (1) を数値積分するものである. 式 (1) の解析解は $\frac{1}{2}\log_e 3$ である. 分割数を 1,2,4 というように $\frac{1}{2}$ ずつ細かくしたときの, 台形公式で求めた積分値と解析解の関係について考察する.

$$\int_0^{\frac{\pi}{6}} \frac{dx}{\cos x} \tag{1}$$

3.2　プログラムの説明

本節では課題 1 で作成したプログラムにおいて, 次に示す 4 つの関数の役割および機能について説明する. なお数学における「関数」とプログラミングにおける「関数」の意味が混在することを防ぐため, 数学における関数を「数学関数」, プログラミングにおける関数を「関数」と表記する.

1. func 関数
2. Trapezoidal 関数

3. TrapezoidalRule 関数

4. main 関数

3.2.1 func 関数

func 関数は数値計算を行う数学関数を定義する関数である. 表 2 に func 関数の機能, 引数, 返り値の 3 つを示す. func 関数は数学関数を定義する関数であるから, 引数 x(double 型) について返り値 $f(x)$ を返す設計になっている.

表 2: func 関数の機能, 引数, 返り値

機能	数学関数を定義する
引数	double x
返り値	double 型

リスト 1 に func 関数のソースコードを示す.func 関数は引数 x について積分を行う数学関数 $f(x) = \frac{1}{\cos x}$ の値を返す. なお cos 関数を扱うためには math.h の include が必要である.

リスト 1: func 関数

```
1 double func(double x){
2     return 1/cos(x);
3 }
```

3.2.2 Trapezoidal 関数

Trapezoidal 関数は数学関数 $f(x)$ において, 与えられた 2 点 a,b における台形公式による数値積分を行う関数である. 2 点 a,b における $f(x)$ の値は $f(a)$,$f(b)$ であるから,a から b までの $f(x)$ の積分は台形公式より式 (2) のように近似できる.

$$\int_a^b f(x)dx \simeq \frac{b-a}{2}(f(a)+f(b)) \tag{2}$$

表 3 に Trapezoidal 関数の機能, 引数, 返り値の 3 つを示す.Trapezoidal 関数は 2 点 a,b における台形公式による数値積分を行う関数であるから,2 点 a,b を引数, 数値積分の結果を返り値とする設計になっている.

表 3: Trapezoidal 関数の機能, 引数, 返り値

機能	2 点 a,b における台形公式による数値積分を返す.
引数	double a,double b
返り値	double 型

リスト 2 に Trapezoidal 関数のソースコードを示す.Trapezoidal 関数は引数 a,b について式 (2) の計算結果を返す.

リスト 2: Trapezoidal 関数

```
1 double Trapezoidal(double a,double b){
2     return (b-a)*(func(a)+func(b))/2;
3 }
```

3.2.3 TrapezoidalRule 関数

TrapezoidalRule 関数は区間 $[a,b]$ を n 個に分割して, 分割した区間のそれぞれに台形公式を適用する関数である. 図 1 に TrapezoidalRule 関数の数値計算にイメージを示す. 図 1 では数学関数 $f(x)$ について区間 $[a,b]$ を n 個に分割し, それぞれの x の値を x_1,x_2,...,x_n としている. 区間 $[x_i,x_{i+1}]$ における台形公式による数値積分は Trapezaoidal 関数によって計算することができるから, 求めたい積分はすべての区間 $[x_1,x_2]$,$[x_2,x_3]$,...,$[x_{n-1},x_n]$ について Trapezaoidal 関数を適用し, この結果の和をとればよい.

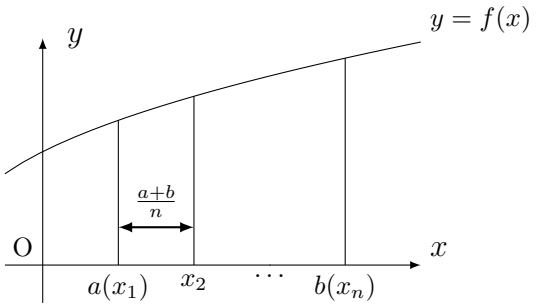

図 1: 合成台形公式のイメージ

表 4 に TrapezoidalRule 関数の機能, 引数, 返り値の 3 つを示す.TrapezoidalRule 関数は 2 点 a,b を n 分割したときの台形公式による数値積分を求める関数であるから,2 点 a,b および分割数 n を引数, 数値積分の結果を返り値とする設計になっている.

表 4: TrapezoidalRule 関数の機能, 引数, 返り値

機能	2 点 a,b を n 分割したときの台形公式による数値積分を返す.
引数	double a,double b,int n
返り値	double 型

リスト 3 に TrapezoidalRule 関数のソースコードを示す. リスト 3 の 2〜5 行目では分割数 n が不正な値 (0 や負) である場合にエラーを表示してプログラムを終了する処理を行ってる. ただし,exit 関数を用いるためには stdlib.h を include する必要がある. そして, リスト 3 の 7〜11 行目では区間 $[a,b]$ を n 分割して, 各区間に Trapezoidal 関数を実行する処理を行っている. 結果は double 型の引数 sum に格納され,return される.

リスト 3: TrapezoidalRule 関数

```
1  double TrapezoidalRule(double a,double b,int n){
2      if(n<=0){
3          printf("Incorrect value of n");
4          exit(0);
5      }
6
7      double sum=0;
8      for(double i=0;i<n;i++){
9          sum+=Trapezoidal(a+i*(b/n),a+(i+1)*(b/n));
10     }
11     return sum;
12 }
```

3.2.4 main 関数

TrapezoidalRule 関数を実行し, 結果を表示する関数として main 関数を作成する. リスト 4 に main 関数のソースコードを示す. リスト 4 の 2 行目では分割数を 1,2,4,... と細かくするときの上限を定義している. リスト 4 では分割数の上限を 256 にしている. そして, リスト 4 の 3 行目では解析解を定義している.

リスト 4 の 5〜11 行目では分割数 1,2,4,...,n_max について TrapezoidalRule 関数を用いて数学関数 $f(x)$ の数値積分を行い, 結果および解析解との誤差の絶対値を表示している. またコメントアウトしている 10 行目は CSV 形式で計算結果を出力するためのフォーマットである.

リスト 4: main1 関数

```
1  int main(int argc,char *argv[]){
2      int n_max = 256;
3      double ans = logf(3)/2;
4
5      double result;
6      for(int i=1;i<=n_max;i*=2){
7          result = TrapezoidalRule(0,M_PI/6,i);
8          printf("n =  %4d  result = %lf  error = %lf\n",i,result,fabs(result-ans));
9          // output format for csv
10         //printf("%d,%lf,%0.15lf\n",i,result,fabs(result-ans));
11     }
12     return 0;
13 }
```

3.3 実行結果

図 2 に課題 1 のプログラムの実行結果を示す.TrapezoidalRule 関数による数値積分の結果は解析解 $\frac{1}{2}\log_e 3 = 0.5493061$ に十分近い値であることがわかる. 注意として, 分割数が 256 のとき誤差が 0 になって完璧な数値解が得られたと思うかもしれないが, これは表示の問題であり実際には表示されている桁よりも下の桁で誤差が存在している. 解析解に収束する様子や誤差の考察については次節で行う.

```
n =    1  result = 0.564099  error = 0.014793
n =    2  result = 0.553084  error = 0.003778
n =    4  result = 0.550256  error = 0.000950
n =    8  result = 0.549544  error = 0.000238
n =   16  result = 0.549366  error = 0.000059
n =   32  result = 0.549321  error = 0.000015
n =   64  result = 0.549310  error = 0.000004
n =  128  result = 0.549307  error = 0.000001
n =  256  result = 0.549306  error = 0.000000
```

図 2: 課題 1 の実行結果

3.4 考察

分割数を変化させたときの収束の様子や誤差について考察を行う. 図 3 に分割数を変化させたときの数値解の収束の様子を示す. ただし分割数は対数軸になっている. 図 3 から分割数を大きくする, すなわち刻み幅を細かくすると数値解が解析解に近づく, つまり誤差が減少していることがわかる.

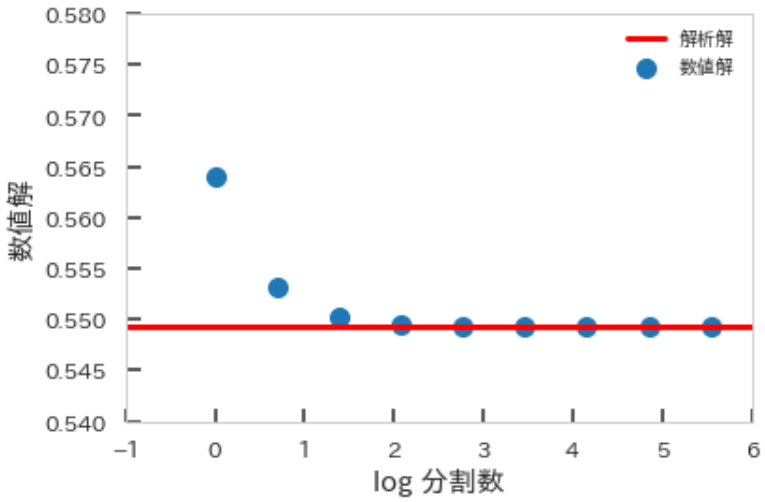

図 3: 数値解の収束の様子 (台形公式)

「わかりやすい数値計算入門」!によれば合成台形公式の誤差は $\mathcal{O}(h^2)$ である.$\mathcal{O}(h^2)$ とは分割数 h^2 が 2 倍になると, 精度が 4 倍になることを意味している. この実験では分割数を 2 倍ずつ大きくしているから,1 つ前の分割数よりも精度が 4 倍よくなると予想できる.

表 5 に 1 つ前の分割数と比較したときの精度の差を示す. 分割数を 2 倍にすると 1 つ前の分割数と比較して, 精度が平均 4.01 倍よくなっている. このことから分割数 h^2 が 2 倍になると, 精度が 4 倍になるという予想が正しいことがわかる.

表 5: 分割数を大きくしたときの誤差の変化

分割数	数値解	誤差	1 つ前との精度の違い (倍)
1	0.564099	0.014793128	-
2	0.553084	0.003778157	3.92
4	0.550256	0.000950031	3.98
8	0.549544	0.000237854	3.99
16	0.549366	5.94782E-05	4.00
32	0.549321	1.48635E-05	4.00
64	0.54931	3.70853E-06	4.01
128	0.549307	9.197E-07	4.03
256	0.549306	2.22487E-07	4.13

このように, 分割数を大きくしたことによって近似の度合いが増加することで減少する誤差を打ち切り誤差という. 打ち切り誤差はコンピュータ内部の数値の扱いによって生じるものではなく, アルゴリズムの近似精度によって生じる誤差である. 台形公式では元の数学関数 $f(x)$ を台形, つまり一次式で近似しているため, 分割数が少ない場合は近似精度が十分ではないと考えられる. このため, 打ち切り誤差が生じる.

打ち切り誤差は分割数を大きくして減少するのであれば, 可能な限り分割数を大きくすれば, より正確な数値解を得られると考えられるが, 分割数を過剰に大きくすると打ち切り誤差とは別に, 丸め誤差の問題が生じる. 丸め誤差とは小数点を含む 10 進数と 2 進数の変換時に生じる誤差である. 例えば,10 進数の 0.6 は 2 進数で表すと 0.100110011... というように無限の桁数が必要である. しかしコンピュータで無限の桁数を

扱うことはできないのでどこかで桁を丸める必要がある.8 ビット目で丸めた場合, 0.10011001 となり, これを 10 進数に戻すと 0.59765626 となり元の 0.6 と誤差が生じる. これが丸め誤差である.

これとは別の誤差の要因として桁落ちが考えられる. 桁落ちとは非常に近い値どうしの引き算を行ったときに有効数字が急激に減少するために生じる誤差のことである. 計算結果の正規化を行う場合, 有効桁に 0 が詰められるわけではない. したがって, 正規化の結果, 有効桁に詰められた 0 でない数字が目立ってしまう. これが桁落ちによって生じる誤差である.

台形公式の近似 (式 (2)) で桁落ちを考慮しなければいけない部分は $b-a$ の部分である. 刻み幅 h を小さくすると TrapezoidalRule 関数で反復的に呼び出す積分区間 $[x_i,x_{i+1}]$ は非常に近い値を持つ. このため,$x_{i+1}-x_i$ の計算において桁落ちが起きる可能性が考えられる.

4 課題 2

本章では課題 2 における課題内容, プログラムの説明, 実験結果, 考察の 4 つについて述べる.

4.1 課題内容

課題 2 の課題内容はシンプソンの公式を用いて式 (3) の数値積分を行うものである. 解析解は 1 である.

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

数値積分は次の 2 点についてプログラムの作成および実行を行い結果を考察する.

1. float 型と double 型のそれぞれで数値積分を行い丸め誤差が現れる刻み幅 h.
2. 刻み幅を $\frac{1}{2}$ 倍ずつ変化させたときの誤差の減少.

4.2 プログラムの説明

本節では課題 2 で作成したプログラムにおいて, 次に示す 4 つの関数の役割および機能について説明する. なお, 課題 2 では double 型および float 型のそれぞれで数値積分を行う関数を作成したが, ここでは double 型のプログラムのみ説明する.float 型のプログラムについては,double の部分を float に変更すれば問題なく実行できる.

1. func 関数
2. Simpson 関数
3. SimpsonRule 関数
4. main 関数

4.2.1 func 関数

func 関数は課題 1 と同様に数値計算を行う数学関数を定義する関数である. したがって関数の機能, 引数, 返り値は表 2 と同じである.
リスト 5 に func 関数のソースコードを示す.func 関数は引数 x について積分を行う数学関数 $f(x)=\sin x$ の値を返す.

リスト 5: func 関数

```
double func(double x){
    return sin(x);
}
```

4.2.2 Simpson 関数

Simpson 関数は数学関数 $f(x)$ において与えられた 2 点 a,b におけるシンプソン法による数値積分を行う関数である. a から b までの $f(x)$ の積分はシンプソン法により式 (4) のように近似できる. ただし,h は積分を行う区間の幅である. すなわち $h = \frac{b-a}{2}$ である.

$$\int_a^b f(x)dx \simeq \frac{h}{3}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right] \tag{4}$$

式 (4) の意味について説明する. 台形公式では 2 点 a,b のおける $f(a)$,$f(b)$ の値のみを用いて数値積分を行っていた (式 (2)) がさらに精度を高めたい. そこで a,b の中点 $\frac{a+b}{2}$ の値を用いて 2 次の近似を行う.3 点における $f(x)$ の値をそれぞれ y_0, y_1, y_2 とする. このとき,$\frac{h}{2}$ を基準とすると,3 点は $(-h, y_0), (0, y_1), (h, y_2)$ と表せる.$f(x)$ を近似する 2 次多項式 $y = a + bx + cx^2$ とおくと式 (5)〜式 (7) の連立方程式が得られる.

$$\begin{cases} a - bh + ch^2 = y_0 & (5) \\ a = y_0 & (6) \\ a + bh + ch^2 = y_2 & (7) \end{cases}$$

式 (5)+ 式 (7)×4+式 (7) を計算すると式 (9) が得られる. これを面積 S の計算に用いることでシンプソン法の式 (4) が得られる.

$$\begin{aligned} (a - bh + ch^2) + 4a + (a + bh + ch^2) &= y_0 + 4y_1 + y_2 & (8) \\ 6a + 2ch^2 &= y_0 + 4y_1 + y_2 & (9) \end{aligned}$$

求めたい面積 S を 2 次多項式の近似で計算すると式 (14) のようにシンプソン法の式 (4) が得られる. 式 (14) の計算において, 式 (13) から式 (14) の計算には式 (9) を用いている.

$$\begin{aligned} S &= \int_{-h}^{h}(a + bx + cx^2)dx & (10) \\ &= 2\int_0^h (a + cx^2)dx & (11) \\ &= 2\left[ax + \frac{cx^3}{3}\right]_0^h & (12) \\ &= \frac{h}{3}(6a + 2ch^2) & (13) \\ &= \frac{h}{3}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right] & (14) \end{aligned}$$

表**??**に Simpson 関数の機能, 引数, 返り値の 3 つを示す.Simpson 関数は 2 点 a,b におけるシンプソン法による数値積分を行う関数であるから,2 点 a,b を引数, 数値積分の結果を返り値とする設計になっている.

表 6: Simpson 関数の機能, 引数, 返り値

機能	2 点 a,b におけるシンプソン法による数値積分を返す.
引数	double a,double b
返り値	double 型

リスト 6 に Simpson 関数のソースコードを示す.Simpson 関数は引数 a,b について式 (4) の計算結果を返す.

リスト 6: Simpson 関数

```
1  double Simpson(double a,double b){
2      double h = (b-a)/2;
3      double c = (a+b)/2;
4      return h*(func(a)+4*func(c)+func(b))/3;
5  }
```

4.2.3 SimpsonRule 関数

SimpsonRule 関数は区間 [a,b] を n 個に分割して, 分割した区間のそれぞれにシンプソン法による数値積分を適用する関数である．区間を分割して積分を行うイメージとしては TrapezoidalRule 関数と同様である.

表 7 に SimpsonRule 関数の機能, 引数, 返り値の 3 つを示す.SimpsonRule 関数は 2 点 a,b を n 分割したときのシンプソン法による数値積分を求める関数であるから,2 点 a,b および分割数 n を引数, 数値積分の結果を返り値とする設計になっている.

表 7: SimpsonRule 関数の機能, 引数, 返り値

機能	2 点 a,b を n 分割したときのシンプソン法による数値積分を返す.
引数	double a,double b,int n
返り値	double 型

リスト 7 に SimpsonRule 関数のソースコードを示す．リスト 7 の 2〜5 行目では分割数 n が不正な値 (0 や負) である場合にエラーを表示してプログラムを終了する処理を行ってる．そして, リスト 7 の 6〜11 行目では区間 [a,b] を n 分割して, 各区間に Simpson 関数を実行する処理を行っている. 結果は double 型の引数 sum に格納され,return される.

リスト 7: SimpsonRule 関数

```
1  double SimpsonRule(double a,double b,int n){
2      if(n<=0){
3          printf("Incorrect value of n");
4          exit(0);
5      }
6      double sum=0;
7      for(double i=0;i<n;i++){
8          sum+=Simpson(a+i*(b/n),a+(i+1)*(b/n));
9      }
10     return sum;
11 }
```

4.2.4 main 関数

SimpsonRule 関数を実行し, 結果を表示する関数として main 関数を作成する. リスト 8 に main 関数のソースコードを示す. リスト 8 の 2 行目では分割数を 1,2,4,. . . と細かくするときの上限を定義している. リスト 8 では分割数の上限を 256 にしている. リスト 8 の 3〜7 行目では分割数 1,2,4,. . .,n_max について SimpsonlRule 関数を用いて数学関数 $f(x)$ の数値積分を行い, 結果 (少数以下 16 桁) および解析解との誤差の絶対値を表示している. ただし解析解 3.141592 はオブジェクト形式マクロで定義している.

リスト 8: main2 関数

```
1 int main(int argc,char *argv[]){
2     int n_max = 256;
3     for(int i=1; i<=n_max; i*=2){
4         double result = SimpsonRule(0,M_PI/2,i);
5         printf("split = %4d   result = %1.16lf   error = %1.16lf\n",
6         i,result,fabs((double)1-result));
7     }
8     return 0;
9 }
```

4.3 実行結果

図 4 に課題 2 のプログラムの double 型における実行結果, 図 5 に float 型における実行結果を示す. float 型では 256 分割,double 型では 65536 分割まで実行した. 図 4 および図 5 において, 分割数 1 から 256 について SimpsonRule 関数を実行できていることがわかる. どちらの実行結果においても数値積分の結果が 1 に収束していることが読み取れる. 解析解に収束する様子や誤差の考察は次節で行う.

```
split =     1   result = 1.0022798774922104   error = 0.0022798774922104
split =     2   result = 1.0001345849741938   error = 0.0001345849741938
split =     4   result = 1.0000082955239677   error = 0.0000082955239677
split =     8   result = 1.0000005166847064   error = 0.0000005166847064
split =    16   result = 1.0000000322650009   error = 0.0000000322650009
split =    32   result = 1.0000000020161288   error = 0.0000000020161288
split =    64   result = 1.0000000001260010   error = 0.0000000001260010
split =   128   result = 1.0000000000078748   error = 0.0000000000078748
split =   256   result = 1.0000000000004921   error = 0.0000000000004921
split =   512   result = 1.0000000000000304   error = 0.0000000000000304
split =  1024   result = 1.0000000000000016   error = 0.0000000000000016
split =  2048   result = 0.9999999999999992   error = 0.0000000000000008
split =  4096   result = 1.0000000000000007   error = 0.0000000000000007
split =  8192   result = 1.0000000000000020   error = 0.0000000000000020
split = 16384   result = 1.0000000000000031   error = 0.0000000000000031
split = 32768   result = 1.0000000000000024   error = 0.0000000000000024
split = 65536   result = 1.0000000000000062   error = 0.0000000000000062
```

図 4: 課題 2 の実行結果 (double 型)

```
split =   1   result = 1.00227987766266   error = 0.00227987766266
split =   2   result = 1.00013458728790   error = 0.00013458728790
split =   4   result = 1.00000834465027   error = 0.00000834465027
split =   8   result = 1.00000059604645   error = 0.00000059604645
split =  16   result = 1.00000011920929   error = 0.00000011920929
split =  32   result = 1.00000011920929   error = 0.00000011920929
split =  64   result = 1.00000011920929   error = 0.00000011920929
split = 128   result = 1.00000011920929   error = 0.00000011920929
split = 256   result = 1.00000000000000   error = 0.00000000000000
```

図 5: 課題 2 の実行結果 (float 型)

4.4 考察

本節では次に示す 2 つの考察について述べる.

4.4.1 丸め誤差の現れる刻み幅 h

丸め誤差とは有限桁の 2 進数で表せない数値を丸めるさいに発生する誤差であった.IEEE754 によれば float 型,double 型の内部表現は次のようになっている.

- float 型 (32 ビット)... 符号部 1 ビット, 指数部 8 ビット, 仮数部 23 ビット
- double 型 (64 ビット)... 符号部 1 ビット, 指数部 11 ビット, 仮数部 52 ビット

float 型の精度は 10 進数で 7 桁,double 型では 15 桁である. すなわち丸め誤差は float 型では 7 桁目,double 型では 15 桁目付近で現れるはずである.

実行結果で実際の丸め誤差を確認する. まず double 型について確認する.double 型の実行結果 (図 4) は小数点以下 16 桁を表示している. 分割数が 4096 のとき, 小数点以下 15 桁に 0 がたっており,16 桁目に 7 がたっている. さらに,4096 より分割数を大きくしても, 数値解が収束していないことが読み取れる. このことから, 分割数が 4096 のときに double 型の丸め誤差が生じると考えられる. これよりも精度が必要な場合は多倍長演算を行う必要がある.

次に float 型について確認する.float 型の実行結果 (図 5) は小数点以下 16 桁を表示している. 分割数が 16 のとき, 小数点以下 6 桁に 0 がたっており,7 桁目に 1 がたっている. さらに, 分割数が 16 から 128 のとき, 数値解がすべて一致している. これらを踏まえると, 分割数が 16 のときに float 型の丸め誤差が生じると考えられる.

4.4.2 刻み幅を $\frac{1}{2}$ ずつ変化させたときの誤差の減少

5 課題 3〜5

本章では課題 3〜5 の課題内容, プログラムの説明, 実行結果, 考察について述べる.

5.1 課題内容

課題 3〜5 では式 (15) に示す微分方程式を 3 つのアルゴリズムで解き, 誤差および精度について考察する. ただし式 (15) の初期条件は $u(0) = 1$ である. 式 (15) の解析解は e^t である. 課題 3 ではオイラー法, 課題 4

ではホイン法, 課題 5 ではルンゲ・クッタ法で微分方程式を解く.

$$\frac{du}{dt} = u \tag{15}$$

5.2 プログラムの説明

本節では課題 3〜5 で作成したプログラムにおいて, 次に示す 5 つの関数の役割および機能について説明する.

1. func 関数
2. Euler 関数
3. Heun 関数
4. RK 関数
5. main 関数

5.2.1 func 関数

func 関数は式 (16) に示すように微分方程式の右辺の数学関数を定義する関数である.

$$\frac{dy}{dx} = f(x, y) \tag{16}$$

表 8 に func 関数の機能, 引数, 返り値の 3 つを示す. func 関数は数学関数を定義する関数であるから, 引数 x,y(double 型) について返り値 $f(x, y)$ を返す設計になっている.

表 8: func 関数の機能, 引数, 返り値

機能	数学関数を定義する
引数	double x,double y
返り値	double 型

リスト 9 に func 関数のソースコードを示す.func 関数は引数 x,y について微分方程式の右辺の数学関数 $f(x) = y$ の値を返す.

リスト 9: func 関数

```
1 double func(double x,double y){
2     return y;
3 }
```

5.2.2 Euler 関数

課題 3 のプログラムとして微分方程式を Euler 法で解く関数として Euler 関数を作成する. 今与えられている微分方程式の情報は式 (16) および初期条件である. この情報のみから, 解析的手法を用いずに数値解を求める方法を考える.

コンピュータでは無限に小さい値を扱うことができないから式 (16) を式 (17) のように近似する.$\frac{\Delta y}{\Delta x}$ は十分小さい値という意味合いだと思ってもらいたい.

$$\frac{dy}{dx} \approx \frac{\Delta y}{\Delta x} = f(x, y) \tag{17}$$

式 (17) を変形すると式 (18) のようになる.

$$\Delta y = f(x, y)\Delta x \tag{18}$$

$\Delta y = y_1 - y_0$,$\Delta x = x_1 - x_0$,$h = x_1 - x_0$ とすると, 式 (18) は式 (19) のように変形できる.

$$y_1 - y_0 \quad = \quad hf(x_0, y_0) \tag{19}$$

これを一般化すると, 式 (26) および式 (21) に示すように x_{i+1},y_{i+1} を反復的に計算することによって数値解を求めることができる. これを Euler 法という.Euler 関数は式 (21) の計算を反復的に行う関数である.

$$x_{i+1} \quad = \quad x_i + h \tag{20}$$

$$y_{i+1} \quad = \quad y_i + hf(x_i, y_i) \tag{21}$$

表 9 に Euler 関数の機能, 引数, 返り値の 3 つを示す. Euler 関数は Euler 法による微分方程式の数値解を求める関数であるから, 初期条件 x_0, y_0 および刻み幅 h を引数とする設計になっている. 結果は標準出力を行うため返り値はない.

表 9: Euler 関数の機能, 引数, 返り値

機能	式 (21) を反復的に計算し, 結果を標準出力に表示する.
引数	double x0,double y0,double h
返り値	なし (void)

リスト 10 に Euler 関数のソースコードを示す. リスト 10 の 4 行目から 10 行目が式 (21) を反復的に計算する部分である. ただしループ回数の上限はオブジェクト形式マクロで MAX_STEP という値を宣言する必要がある.

リスト 10: Euler 関数

```
1  void Euler(double x0,double y0,double h){
2      double x=x0,y=y0;
3      double xp,yp;
4      for(int i=0;i<MAX_STEP;i++){
5          xp = x+h;
6          yp = y+h*func(x,y);
7          printf("t=%02d   x=%lf,y=%lf\n",i+1,xp,yp);
8          x=xp;
9          y=yp;
10     }
11 }
```

5.2.3 Heun 関数

課題 4 のプログラムとして微分方程式を Heun 法で解く関数である Heun 関数を作成する.Heun 法は Euler 法を改善し, 精度を向上させた方法である. 式 (25) に Heun 法の計算式を示す. Euler 法では $f(x_i, y_i)$ のみ

を用いて数値解を求めたが,Heun 法では $f(x_i, y_i)$ と $f(x_i + h, y_i + hf(x_i, y_i))$ の平均を用いて数値解を計算している.

$$x_{i+1} = x_i + h \tag{22}$$
$$k_1 = hf(x_i, y_i) \tag{23}$$
$$k_2 = h(x_i + h, y_i + k_1) \tag{24}$$
$$y_{i+1} = y_i + \frac{1}{2}(k_1 + k_2) \tag{25}$$

Heun 関数は,Euler 関数と微分方程式を数値解で求める内部のアルゴリズムが異なるだけである. このため, 関数設計 (引数, 返り値) は表 9 と同じである.

リスト 11 に Heun 関数のソースコードを示す. リスト 11 の 4 行目から 11 行目が反復的に式 (25) を計算している部分である.

リスト 11: Heun 関数

```
1  void Heun(double x0,double y0,double h){
2      double x=x0,y=y0;
3      double xp,yp,k1,k2;
4      for(int i=0;i<MAX_STEP;i++){
5          xp = x+h;
6          k1 = h*func(x,y);
7          k2 = h*func(x+h,y+k1);
8          yp = y+(k1+k2)/2;
9          printf("t=%02d   x=%lf,y=%lf\n",i+1,xp,yp);
10         x=xp;
11         y=yp;
12     }
13 }
```

5.2.4 RK 関数

課題 5 のプログラムとして微分方程式をルンゲ・クッタ法で解く関数である RK 関数を作成する. ルンゲ・クッタ法は一般に微分方程式の数値解を求めるのに用いられる 4 次近似の方法である.Heun 法で 2 次近似を行ったものと考え方は同じである. 式 (31) にルンゲ・クッタ法の計算式を示す. 式 (31) において,k_2,k_3 は 2 倍の重みで扱われている.

$$x_{i+1} = x_i + h \tag{26}$$
$$k_1 = hf(x_i, y_i) \tag{27}$$
$$k_2 = h(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}) \tag{28}$$
$$k_3 = h(x_i + \frac{h}{2}, y_i + \frac{k_2}{2}) \tag{29}$$
$$k_4 = hf(x_i + h, y_i + k_3) \tag{30}$$
$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{31}$$

RK 関数の関数設計も Euler 関数,Heun 関数の内部のアルゴリズムが異なるだけであるから, 表 9 と同じである.

リスト 12 に Heun 関数のソースコードを示す. リスト 12 の 4 行目から 20 行目が反復的に式 (31) を計算している部分である.

リスト 12: RK 関数

```
void RK(double x0,double y0,double h){
    double x=x0,y=y0;
    double xp,yp,k1,k2,k3,k4;
    for(int i=0;i<MAX_STEP;i++){
        xp = x+h;
        k1 = h*func(x,y);
        k2 = h*func(x+h/2,y+k1/2);
        k3 = h*func(x+h/2,y+k2/2);
        k4 = h*func(x+h,y+k3);
        yp = y+(k1+2*k2+2*k3+k4)/6;
        printf("t=%02d   x=%lf,y=%lf\n",i+1,xp,yp);
        /* print k1,k2,k3,k4
        printf("k1=%lf\n",k1);
        printf("k2=%lf\n",k2);
        printf("k3=%lf\n",k3);
        printf("k4=%lf\n",k4);
        */
        x=xp;
        y=yp;
    }
}
```

5.2.5 main 関数

Euler 関数,Heun 関数,RK 関数を実行する関数としてメイン関数を作成する. リスト 13 に main 関数のソースコードを示す. リスト 13 では, 初期条件および刻み幅 h の定義をしている. そして 6 行目から 14 行目で Euler 関数,Heun 関数,RK 関数を実行している.

リスト 13: main 関数

```
int main(int argc,char *argv[]){
    double x0 = 0;
    double y0 = 1;
    double h = 0.1;

    /* format for stdout*/
    printf("-----setting-----\n");
    printf("h=%lf\nt=00   x=%lf,y=%lf\n",h,x0,y0);
    printf("\n-----Euler method-----\n");
    Euler(x0,y0,h);
    printf("\n-----Heun method-----\n");
    Heun(x0,y0,h);
    printf("\n-----Runge-kutta method-----\n");
    RK(x0,y0,h);

    return 0;
}
```

5.3 実行結果

表 10〜表 12 に刻み幅 h を次に示す 3 つの値に変化させたときの実行結果を示す.

1. MAX_STEP = 10,h=0.1
2. MAX_STEP = 2,h=0.5
3. MAX_STEP = 20,h=0.05

表 10: MAX_STEP = 10,h=0.1 のときの実行結果

step	t	u(Euler)	u(Heun)	u(RK)
0	0	1	1	1.000000000000000
1	0.1	1.1	1.105	1.105170833333330
2	0.2	1.21	1.221025	1.221402570850690
3	0.3	1.331	1.349233	1.349858497062530
4	0.4	1.4641	1.490902	1.491824240080680
5	0.5	1.61051	1.647447	1.648720638596830
6	0.6	1.771561	1.820429	1.822117962091930
7	0.7	1.948717	2.011574	2.013751626596770
8	0.8	2.143589	2.222789	2.225539563292310
9	0.9	2.357948	2.456182	2.459601413780070
10	1	2.593742	2.714081	2.718279744135160

表 11: MAX_STEP = 2,h=0.5 のときの実行結果

step	t	u(Euler)	u(Heun)	u(RK)
0	0	1	1	1.000000000000000
1	0.5	1.5	1.625	1.648437500000000
2	1	2.25	2.640625	2.717346191406250

表 12: MAX_STEP = 20,h=0.05 のときの実行結果

step	t	u(Euler)	u(Heun)	u(RK)
0	0	1	1	1.000000000000000
1	0.05	1.05	1.05125	1.051271093750000
2	0.1	1.1025	1.105127	1.105170912554320
3	0.15	1.157625	1.161764	1.161834234021660
4	0.2	1.215506	1.221305	1.221402745956150
5	0.25	1.276282	1.283897	1.284025400650570
6	0.3	1.340096	1.349696	1.349858787344710
7	0.35	1.4071	1.418868	1.419067523779920
8	0.4	1.477455	1.491585	1.491824667829220
9	0.45	1.551328	1.568029	1.568312150232060
10	0.5	1.628895	1.64839	1.648721229515870
11	0.55	1.710339	1.73287	1.733252970241990
12	0.6	1.795856	1.82168	1.822118745771740
13	0.65	1.885649	1.915041	1.915540766809830
14	0.7	1.979932	2.013187	2.013752637046890
15	0.75	2.078928	2.116363	2.116999937290230
16	0.8	2.182875	2.224826	2.225540839543780
17	0.85	2.292018	2.338849	2.339646752572480
18	0.9	2.406619	2.458715	2.459603000565510
19	0.95	2.52695	2.584724	2.585709536595280
20	1	2.653298	2.717191	2.718281692656330

5.4 考察

本節では各手法における誤差, 精度の様子, および誤差の考察について述べる.

5.4.1 誤差, 精度の様子

各手法について, 刻み幅 h を変化させたときの数値解を考察する. まず Euler 法による数値解をプロットしてみる. 図 6 に刻み幅 h を 0.1,0.5,0.05 にしたときの Euler 法の数値解のプロットを示す.t が小さいうちはどの刻み幅でも誤差が小さいことが読み取れる. 一方で $t = 1$ 付近では数値解と解析解の誤差が大きく, 特に刻み幅 $h = 0.5$ の数値解の誤差大きく目立つことがわかる.$h = 0.5, t = 1$ のときの Euler 法の数値解は, 表 11 より 2.25 である. 解析解 $e = 2.7181...$ であるから 0.468 という非常に大きい誤差があることがわかる. 一方で最も正確な $h = 0.05, t = 1$ のときの Euler 法による数値解は, 表 12 より 2.653298 であることがわかる. 解析解 $e = 2.7181...$ との誤差は 0.065 である. 同条件でのルンゲ・クッタ法の数値解が 2.718282 と比較すると精度が不十分であることがわかる.

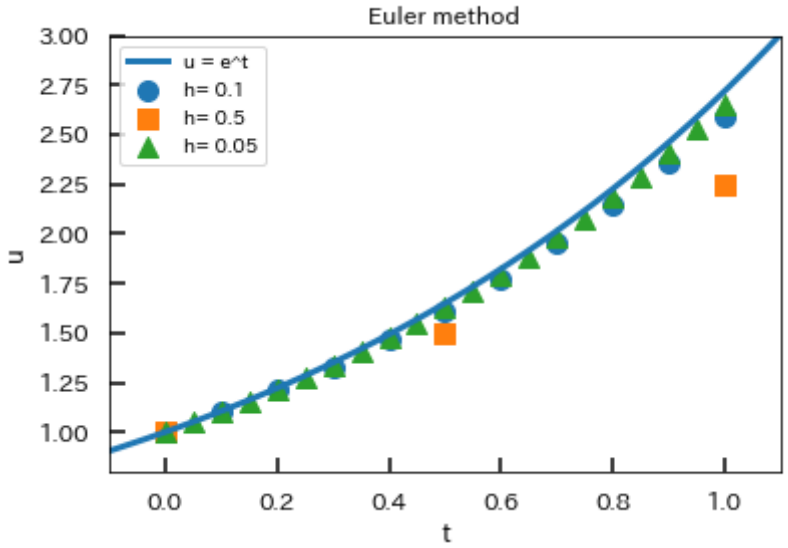

図 6: 刻み幅 h を変化させたときの数値解の様子 (Euler 法)

次に Heun 法による数値解をプロットして, 様子を考察する. 図 7 に刻み幅 h を 0.1,0.5,0.05 にしたときの Euler 法の数値解のプロットを示す. 図 6 と図 7 を比較すると Heun 法のほうがどの刻み幅 h でも数値解の誤差が小さいことが読み取れる. Euler 法で誤差が大きかった $h = 0.5, t = 1$ の数値解は表 11 より 2.640625 であることが読み取れる. 解析解との差は 0.078 であり Euler 法と比較すると 0.4 も精度が上がっていることがわかる. また,$h = 0.05, t = 1$ のときは, 表 12 より数値解は 2.717191 であることが読み取れる. 同条件の Euler 法と比較すると,0.06 精度が上がっていることが読み取れる. これらを踏まえると,Euler 法よりも 2 次の近似を行った Heun 法のほうが精度がよいことがわかる.

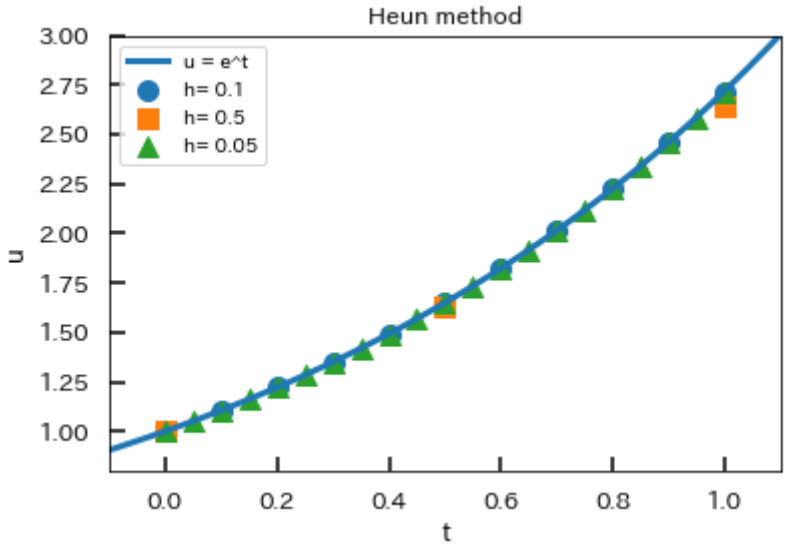

図 7: 刻み幅 h を変化させたときの数値解の様子 (Heun 法)

最後にルンゲ・クッタ法による数値解をプロットして様子を考察する. 図 8 に刻み幅 h を 0.1,0.5,0.05 にしたときのルンゲ・クッタ法の数値解のプロットを示す. 図 8 において数値解は解曲線上にのっていること

が読み取れる. このことから Euler 法,Heun 法と比較してルンゲ・クッタ法はより精度が高いことが考えられる. 実際,$h = 0.05, t = 1$ のときの数値解は 2.718282 であり, 解析解 $e = 2.7181828...$ と小数点以下 6 桁は一致していることがわかる.

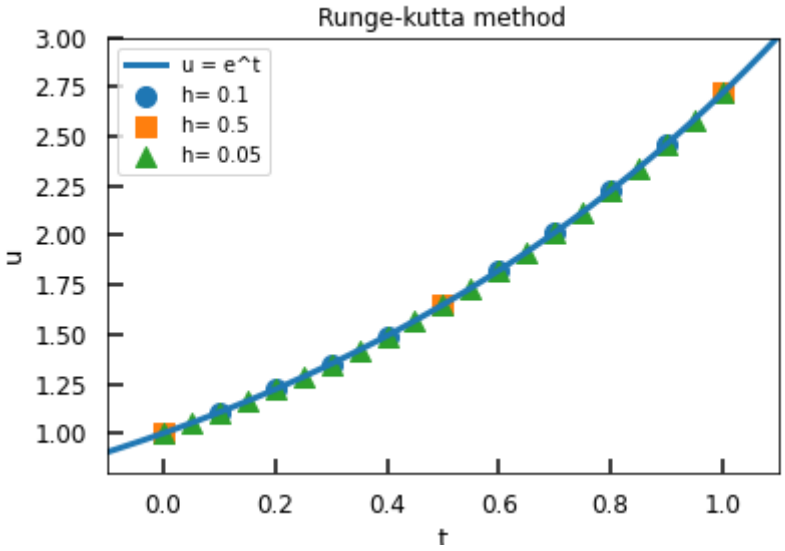

図 8: 刻み幅 h を変化させたときの数値解の様子 (ルンゲ・クッタ法)

5.4.2 誤差の考察

「わかりやすい数値計算入門」!によれば Euler 法,Heun 法, ルンゲ・クッタ法の誤差のオーダーは表 13 のようになっている. ここで局所離散化誤差とは各ステップで生じる誤差のことであり, 大域離散化誤差は局所離散化誤差が蓄積した誤差のことである. 今考えている誤差は大域離散化誤差である.

表 13: ルンゲ・クッタ型公式の誤差

公式	次数	局所誤差	大域誤差
Euler 法	1	$\mathcal{O}(h^2)$	$\mathcal{O}(h)$
Heun 法	2	$\mathcal{O}(h^3)$	$\mathcal{O}(h^2)$
ルンゲ・クッタ法	4	$\mathcal{O}(h^5)$	$\mathcal{O}(h^4)$

大域離散化誤差が表 13 に従っているか確認する. まず Euler 法の場合について確認する. この実験では刻み幅 h を 0.5,0.1,0.05 のように変化させているから刻み幅 h =0.5 の数値解と刻み幅 h =0.05 の数値解では精度に 10 倍の違いがあり, 刻み幅 h0.1 の数値解と刻み幅 h0.05 の数値解では精度に 2 倍の違いがあると予想できる.

表 14 に $t = 0.5$ および $t = 1$ のときの解析解と数値解との誤差を示す. ただし $e^{0.5} = 1.6487...$, $e = 2.7182...$ である. 表 14 から刻み幅 h =0.5 の数値解と刻み幅 h =0.05 の数値解では精度は平均 7.25 倍よくなっていることが計算できる. これは理論値である 10 倍の精度になっているとは言えないが,$\mathcal{O}(h)$ から大きく外れた精度にはなっていない. 刻み幅 h =0.1 の数値解と刻み幅 h =0.05 の数値解では精度が平均 1.91 倍よくなっていることが計算できる. これは理論値である 2 倍に合致した結果である. 以上のことから Euler 法の大域離散化誤差は $\mathcal{O}(h)$ になっていることが確認できたと考える.

表 14: 数値解と解析解の誤差 (Euler 法)

	$h = 0.5$	$h = 0.1$	$h = 0.05$
$t = 0.5$	0.149	0.038	0.020
$t = 1.0$	0.458	0.125	0.065

次に Heun 法の場合について確認する.Heun 法の場合も $t = 0.5, t = 1.0$ で検証する. 表 15 に $t = 0.5$ および $t = 1$ のときの解析解と数値解との誤差を示す. Heun 法の大域離散化誤差は $\mathcal{O}(h^2)$ であるから, 刻み幅 h =0.5 の数値解と刻み幅 h =0.05 の数値解では精度に 100 倍の違いがあり, 刻み幅 h0.1 の数値解と刻み幅 h0.05 の数値解では精度に 4 倍の違いがあると予想できる.
表 14 から刻み幅 h =0.5 の数値解と刻み幅 h =0.05 の数値解では精度は平均 71.2 倍よくなっていることが計算できる. これは理論値である 100 倍の精度になっているとは言えないが,$\mathcal{O}(h^2)$ から大きく外れた精度にはなっていない. 刻み幅 h =0.1 の数値解と刻み幅 h =0.05 の数値解では精度が平均 7.55 倍よくなっていることが計算できる. これは理論値である 4 倍の精度になっているとは言えないが,$\mathcal{O}(h)$ から大きく外れた精度にはなっていない. これらから, 理論値の精度には達していないが,Heun 法の大域離散化誤差は $\mathcal{O}(h^2)$ になっていることが確認できたと考える.

表 15: 数値解と解析解の誤差 (heun 法)

	$h = 0.5$	$h = 0.1$	$h = 0.05$
$t = 0.5$	0.02371	0.00123	0.00033
$t = 1.0$	0.0777	0.0042	0.0011

最後ににルンゲ・クッタ法の場合について確認する. ルンゲ・クッタ法も $t = 0.5, t = 1.0$ で検証する. 表 16 に $t = 0.5$ および $t = 1$ のときの解析解と数値解との誤差を示す. ルンゲ・クッタ法の大域離散化誤差は $\mathcal{O}(h^4)$ であるから, 刻み幅 h =0.5 の数値解と刻み幅 h =0.05 の数値解では精度に 10^4 倍の違いがあり, 刻み幅 h0.1 の数値解と刻み幅 h0.05 の数値解では精度に $2^4 = 16$ 倍の違いがあると予想できる.
表 16 から刻み幅 h =0.5 の数値解と刻み幅 h =0.05 の数値解では精度は平均 6.5×10^3 倍よくなっていることが計算できる. これは理論値である 10^4 倍の精度になっているとは言えないが,$\mathcal{O}(h^4)$ から大きく外れた精度にはなっていない. 刻み幅 h =0.1 の数値解と刻み幅 h =0.05 の数値解では精度が平均 15 倍よくなっていることが計算できる. これは理論値である 16 倍にほぼ等しい. これらから, 理論値の精度には達していないが, ルンゲ・クッタ法の大域離散化誤差は $\mathcal{O}(h^4)$ になっていることが確認できたと考える.

表 16: 数値解と解析解の誤差 (ルンゲ・クッタ法)

	$h = 0.5$	$h = 0.1$	$h = 0.05$
$t = 0.5$	2.8×10^{-4}	6.3×10^{-7}	4.1×10^{-8}
$t = 1.0$	9.4×10^{-4}	2.1×10^{-6}	1.4×10^{-7}

参考文献

[1] 伊藤祥一, 多倍長演算 3J アルゴリズムとデータ構造 後期,2019 年 9 月 4 日