数理工学実験
テーマ:常微分方程式の数値解法

田中風帆 (1029321151)
実施場所:自宅

実施:2021 年 10 月 11 日
提出:2021 年 10 月 21 日

1　概要

このレポートでは、常微分方程式の数値解法に関する問題を解き、議論を行う。課題 1 では身の回りにある現象を常微分方程式の初期値問題として定式化する。課題 2 では、4 次のアダムスバッシュホース法とアダムスムルトン法を構成し、さらにそれらを用いて予測子修正子法を構成する。課題 3 では具体的な常微分方程式の初期値問題を数値的にとき、解析解と比較する。課題 4 では、常微分方程式の初期値問題を複数の手法で解いた際、安定性がどのように変わるかを数値的に確認する。課題 5 では、新たに作られた数値計算手法がある特定の初期値問題を解くことができないことを示し、実際に数値計算で確認する。課題 6 では、生物の個体数の増減に関する数理モデルに対して数値解析を行い、解の振る舞いを調べる。課題 7 では、前進オイラー法とルンゲクッタ法を用いてローレンツ方程式の初期値問題を数値解析する。

2　課題 1

ここでは、身の回りにある現象を常微分方程式の初期値問題として定式化した。今回は、RL 直列回路を取り上げた。
RL 直列回路の状態方程式を定式化する。電圧を E, 抵抗を R, コイルを L とし、流れる電流を i とする。また、t を時間を表す独立変数とする。t が 0 の時、電流は流れていないものとする。この時、状態方程式と初期条件は以下のように記述できる。

$$L\frac{di}{dt} + Ri = E \tag{1}$$

$$i(0) = 0 \tag{2}$$

3　課題 2

ここでは、4 次のアダムスバッシュホース法とアダムスムルトン法を構成し、さらにそれらを用いて予測子修正子法を構成する。
まず、4 次のアダムスバッシュホース法を構築する。式 $u_n - u_{n-1} = \int_{t_{n-1}}^{t_n} f(\tau, u(\tau))d\tau$ において、f を τ に関する多項式 p で近似する。p は 3 次の Lagrange 補間によって構築する。この時、$p(\tau) = \sum_{k=n-4}^{n-1} \prod_{j=n-4, j\neq k}^{n-1} \frac{\tau - t_j}{t_k - t_j} f_k$(ただしここで $f_n = f(t_n, u_n)$) となる。$t_n - t_{n-1} = \Delta t = h$ を用いて変形すると、$p(\tau) = \frac{(\tau - t_{n-4})(\tau - t_{n-3})(\tau - t_{n-2})}{6h^3} f_{n-1} - \frac{(\tau - t_{n-4})(\tau - t_{n-3})(\tau - t_{n-1})}{2h^3} f_{n-2} + \frac{(\tau - t_{n-4})(\tau - t_{n-2})(\tau - t_{n-1})}{2h^3} f_{n-3} - \frac{(\tau - t_{n-3})(\tau - t_{n-2})(\tau - t_{n-1})}{6h^3} f_{n-4}$ となる。これを積分の式 $u_n - u_{n-1} = \int_{t_{n-1}}^{t_n} f(\tau, u(\tau))d\tau$ に代入し、解は $u_n = u_{n-1} + \Delta t(-\frac{9}{24} f_{n-4} + \frac{37}{24} f_{n-3} - \frac{59}{24} f_{n-2} + \frac{55}{24} f_{n-1})$ となる。
次に、4 次のアダムスムルトン法を構成する。
まず、式 $u_n - u_{n-1} = \int_{t_{n-1}}^{t_n} f(\tau, u(\tau))d\tau$ において、f を τ に関する多項式 p

で近似する。p は 3 次の Lagrange 補間によって構築する。この時、$p(\tau) = \sum_{k=n-3}^{n} \prod_{j=n-3, j\neq k}^{n} \frac{\tau - t_j}{t_k - t_j} f_k$(ただしここで $f_n = f(t_n, u_n)$) となる。$t_n - t_{n-1} = \Delta t = h$ を用いて変形すると、$p(\tau) = \frac{(\tau - t_{n-3})(\tau - t_{n-2})(\tau - t_{n-1})}{6h^3} f_n - \frac{(\tau - t_{n-3})(\tau - t_{n-2})(\tau - t_n)}{2h^3} f_{n-1} + \frac{(\tau - t_{n-3})(\tau - t_{n-1})(\tau - t_n)}{2h^3} f_{n-2} - \frac{(\tau - t_{n-2})(\tau - t_{n-1})(\tau - t_n)}{6h^3} f_{n-3}$ となる。これを積分の式に代入し、解は $u_n = u_{n-1} + \Delta t(\frac{1}{24} f_{n-3} - \frac{5}{24} f_{n-2} + \frac{19}{24} f_{n-1} + \frac{9}{24} f_n)$ となる。
最後に、これら二つを用いて予測子修正法を構成する。アダムスバッシュホース法で u_n の予測値 $\tilde{u}$ を求めると、上の導出より $\tilde{u_n} = u_{n-1} + \Delta t(-\frac{9}{24} f_{n-4} + \frac{37}{24} f_{n-3} - \frac{59}{24} f_{n-2} + \frac{55}{24} f_{n-1})$ となる。これを上のアダムスムルトン法 $u_n = u_{n-1} + \Delta t(\frac{1}{24} f_{n-3} - \frac{5}{24} f_{n-2} + \frac{19}{24} f_{n-1} + \frac{9}{24} f_n)$ における f_n を求めるのに必要な u_n として用いることで、4 次の予測子、修正子法が得られる。

4 課題 3

ここでは、具体的な常微分方程式の初期値問題 $u' = u, u(0) = 1 ...(*)$ を $t \in [0, 1]$ の範囲内で数値的に解き、解析解である $u = \exp(t)$ と比較する。
数値的に解く際、時刻のステップ幅を $\Delta t = \frac{1}{2^i}$、離散化した時間変数を $t_n = n\Delta t$ で定義する。また、ステップ幅が $\Delta t = \frac{1}{2^i}$ の条件下で得られた数値解を u_n^i と表す。ここで、i の値を 1,2,...,i_{max} の範囲で変更した計算を行い、それらの結果を比較することで精度を確認する。今回、i_{max} の値は 8 とした。比較の際は、$t = 1$ における数値解 u_N^i と解析解の差である $E^i = |u_N^i - u(1)|$ を用いる。p を計算手法の次数とし、$E_r^i = \frac{E^i}{E^{i-1}}$(i=2,...,$i_max$) とすると、$E_r^i$ は i の増加とともに $\frac{1}{2^p}$ に近づくはずであるため、コードを実行した際に実際そうなるかどうかを見ることで精度を確認できる。以下、5 種類の方法で $(*)$ を解き E_r^i の値を求め表にまとめる。
まず、前進オイラー法を用いて解く。結果は以下のようになった。倍精度で小数点以下第 8 桁まで求めてある。前進オイラー法は 1 段法なので、理論上は $\frac{1}{2}$ に近づくはずである。実際、E_r^i の値は $\frac{1}{2}$ に近づいていることが確認できる。

コードは以下である。

コード 1: 前進オイラー法のコード

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  double dudt( double u){
6      return u;
7  }
8
9  // オイラー法
10 double solve(double u0, double t0, double tn, int n){
11     int i;
12     double u, t, dt;
```

表 1: 前進オイラー法を用いた際の E_r^i

i	E_r^i
$i=2$	0.59125843
$i=3$	0.55077922
$i=4$	0.52691637
$i=5$	0.51387663
$i=6$	0.50704806
$i=7$	0.50355215
$i=8$	0.50178319

```
13      u = u0;//u の初期値
14      t = t0;//t の初期値
15      dt = (tn − t0) /n;
16
17      // 漸化式を計算
18      for ( i=1; i <= n ; i++){
19          u += dudt(u) ∗ dt;
20          t = t0 + i∗dt;
21      }
22      return u;
23  }
24  //実行
25  int main(){
26      double E = 0;
27      for(int i=1; i<= 8; i++){
28          double u = solve(1, 0, 1, (int)pow(2,i));
29          if(i >= 2){//E_r を求められる場合は求める
30              printf("%.8f␣",fabs(exp(1)−u)/E);
31          }
32          E = fabs(exp(1)−u);
33      }
34  }
```

以下、コードの説明を行う。関数 solve 内にて、前進オイラー法を定義する。引数は、u_0 を初期値とし、時刻 t_0 から t_n までを n ステップに離散化するということ表す。関数内では、ステップ幅である dt を定義し、更新を n 回にわたり繰り返す。実行部では、E^i の値が定義されていない $i=1$ の場合を除き、一度オイラー法を実行するごとに E^i および E_r^i の値を求める。
次に、2 次のアダムスバッシュホース法で同様の解析を行う。以下が結果である。理論上、E_r^i の値は $\frac{1}{2^2}=\frac{1}{4}$ に近づくはずであり、実際そうなっているのが確認できる。コードは以下である。

コード 2: 2 次のアダムスバッシュホース法のコード

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  double dudt( double u){
6      return u;
```

表 2: 2 次のアダムスバッシュホース法を用いた際の E_r^i

i	E_r^i
$i=2$	0.51077308
$i=3$	0.33171271
$i=4$	0.28243881
$i=5$	0.26432037
$i=6$	0.25670114
$i=7$	0.25323734
$i=8$	0.25159053

```
}

//2 次のアダムスバッシュホース法
double solve(double u0, double t0, double tn, int n){
    double u, t, dt, dudt_1, dudt_2;

    u = u0;//u の初期値
    t = t0;//t の初期値
    dt = (tn − t0) /n;//刻み幅
    dudt_2 = 0;

    // 漸化式を計算
    for (int i=1; i <= n ; i++){
        t += dt;
        dudt_1 = dudt(u);
        if(i < 2){//初期値が足りない時は解析解で計算
            u = exp(t);
        }
        else{
            u += dt ∗ (3∗dudt_1 − dudt_2)/2.0;
        }
        dudt_2 = dudt_1;//変数を次の段階に更新
    }
    return u;
}

//実行
int main(){
    double E = 0;
    for(int i=1; i<= 8; i++){
        double u = solve(1, 0, 1, (int)pow(2,i));
        if(i >= 2){//E_r が定義できる時、計算
            printf("%.8f␣",fabs(exp(1)−u)/E);
        }
        E = fabs(exp(1)−u);
    }
}
```

以下、コードの説明を行う。構造はオイラー法のコードと同じである。関数 solve 内では、アダムスバッシュホース法を定義している。2 次のアダムスバッシュホース法は、$u_{n-1} = u_{n-1} + \frac{\Delta t}{2}(3f_{n-1} - f_{n_2})$ で表される。また、はじめの 1 ステップでは初期値が不足しているため、u_1 の値は解析解の値であ

る $\exp(dt)$ とする (ただし dt はステップ幅)。一度ステップが進むたびに前の u_{n-1} の値を u_{n-2} に代入している。
次に、3 次のアダムスバッシュホース法同様の解析を行う。以下が結果である。理論上、E_r^i の値は $\frac{1}{2^3} = \frac{1}{8}$ に近づくはずであり、実際そうなっているのが確認できる。コードは以下である。

表 3: 3 次のアダムスバッシュホース法を用いた際の E_r^i

i	E_r^i
$i = 2$	inf
$i = 3$	0.22309560
$i = 4$	0.15660176
$i = 5$	0.13856485
$i = 6$	0.13135567
$i = 7$	0.12808423
$i = 8$	0.12652019

コード 3: 3 次のアダムスバッシュホース法のコード

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  double dudt( double u){
6      return u;
7  }
8
9  // 次のアダムスバッシュ法 3
10 double solve(double u0, double t0, double tn, int n){
11     int i;
12     double u, t, dt, dudt_1, dudt_2, dudt_3;
13
14     u = u0;
15     t = t0;
16     dt = (tn - t0) /n;
17     dudt_2 = 0;
18     dudt_3 = 0;
19
20     // 漸化式を計算
21     for ( i=1; i <= n ; i++){
22         dudt_1 = dudt(u);
23         t += dt;
24         if(i < 3){//初期値が揃っていない時は解析解で計算
25             u = exp(t);
26         }
27         else{
28             u += dt * (5*dudt_3-16*dudt_2+23*dudt_1)/12.0;
29         }
30         dudt_3 = dudt_2; dudt_2 = dudt_1;
31     }
32     return u;
33 }
34
35 int main(){
```

```
36      double E = 0;
37      for(int i=1; i<= 8; i++){
38          double u = solve(1, 0, 1, (int)pow(2,i));
39          if(i >= 2){
40              printf("%.8f␣",fabs(exp(1)−u)/E);
41          }
42          E = fabs(exp(1)−u);
43      }
44  }
```

以下、コードの説明を行う。構造はオイラー法のコードと同じである。関数 solve 内では、アダムスバッシュホース法を定義している。3 次のアダムスバッシュホース法は、$u_{n-1} = u_{n-1} + \frac{\Delta t}{12}(23f_{n-1} - 16f_{n_2} + 5f_{n-3})$ で表される。また、はじめの 2 ステップでは初期値が不足しているため、u_1, u_2 の値はそれぞれ解析解の値である $\exp(dt), \exp(2dt)$ とする (ただし dt はステップ幅)。一度ステップが進むたびに前の u_{n-2} の値を u_{n-3} に、u_{n-1} の値を u_{n-2} に代入している。

次に、ホイン法で解析を行う。以下が結果である。ホイン法は 2 次であるため、理論上 E_r^i の値は $\frac{1}{2^2} = \frac{1}{4}$ に近づくはずであり、実際そうなっているのが確認できる。以下がコードである。

表 4: ホイン法を用いた際の E_r^i

i	E_r^i
$i = 2$	0.30166231
$i = 3$	0.27493178
$i = 4$	0.26213530
$i = 5$	0.25596929
$i = 6$	0.25295791
$i = 7$	0.25147200
$i = 8$	0.25073422

コード 4: ホイン法のコード

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  double dudt( double u){
6      return u;
7  }
8
9  //ホイン法
10 double solve(double u0, double t0, double tn, int n){
11     int i;
12     double u, t, dt, k1;
13     u = u0;
14     t = t0;
15     dt = (tn − t0) /n;
16
```

```
17      // 漸化式を計算
18      for ( i=1; i <= n ; i++){
19          k1 = dudt(u);
20          double u_astalisk = u + k1 * dt;
21          u += (k1 + dudt(u_astalisk))*dt/2;
22          t += dt;
23      }
24      return u;
25  }
26
27  int main(){
28      double E = 0;
29      for(int i=1; i<= 8; i++){
30          double u = solve(1, 0, 1, (int)pow(2,i));
31          if(i >= 2){
32              printf("%.8f␣",fabs(exp(1)−u)/E);
33          }
34          E = fabs(exp(1)−u);
35      }
36  }
```

以下、コードの説明を行う。solve 関数内ではホイン法を定義する。ホイン法は、$u_n^* = u_{n-1} + f_{n-1}\Delta t$ を用いて、式 $u_n = u_{n-1} + \frac{\Delta t}{2}(f_{n-1} + f(t_n, u_n^*))$ により u_n を更新するアルゴリズムである。コードの構成は今までのアルゴリズムと同様である。

次に、4 次のルンゲクッタ法を用いて解析を行う。以下が結果である。これは 4 次であるため、理論上 E_r^i の値は $\frac{1}{2^4} = \frac{1}{16}$ に近づくはずであり、実際そうなっているのが確認できる。コードは以下である。

表 5: 4 次のルンゲクッタ法を用いた際の E_r^i

i	E_r^i
$i = 2$	0.07683456
$i = 3$	0.06932944
$i = 4$	0.06583380
$i = 5$	0.06414711
$i = 6$	0.06331867
$i = 7$	0.06290955
$i = 8$	0.06274067

コード 5: 4 次のルンゲクッタ法のコード

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <math.h>
4
5   double dudt( double u){
6       return u;
7   }
8
9   //4 次のルンゲクッタ法
10  double solve(double u0, double t0, double tn, int n){
```

```
11      int i;
12      double u, t, dt;
13      u = u0;
14      t = t0;
15      dt = (tn − t0) /n;
16      double f1, f2, f3, f4;
17      // 漸化式を計算
18      for ( i=1; i <= n ; i++){
19          f1 = dudt(u);
20          f2 = dudt(u + f1∗dt/2);
21          f3 = dudt(u + f2∗dt/2);
22          f4 = dudt(u + f3∗dt);
23          u += (f1 + 2∗f2 + 2∗f3 + f4)∗dt/6;
24          t += dt;
25      }
26      return u;
27  }
28  //実行
29  int main(){
30      double E = 0;
31      for(int i=1; i<= 8; i++){
32          double u = solve(1, 0, 1, (int)pow(2,i));
33          if(i >= 2){
34              printf("%.8f␣",fabs(exp(1)−u)/E);
35          }
36          E = fabs(exp(1)−u);
37      }
38  }
```

以下、コードの説明を行う。コード全体の構成は今まで述べてきたアルゴリズムと同じである。4 次のルンゲクッタ法は、$F_1 = f(t_{n-1}, u_{n-1}), F_2 = f(t_{n-1} + \frac{\Delta t}{2}, u_{n-1} + F_1\frac{\Delta t}{2}), F_3 = f(t_{n-1} + \frac{\Delta t}{2}, u_{n-1} + F_2\frac{\Delta t}{2}), F_4 = f(t_{n-1}, u_{n-1} + F_3\Delta t)$ で表される。これを $t = n$ となるまで計算する。

5 課題 4

ここでは、$t \in [0, \infty]$ の範囲内において、常微分方程式の初期値問題 $u' = -\alpha u + \beta, u(0) = u_0(\alpha > 0)$ を理論的、数値的に解き、安定性を確認する。今回は $\alpha = 10, \beta = 1, u_0 = 1$、つまり $u' = -10u + 1, u(0) = 1$ を解くこととし、手法としてクランクニコルソン法、予測子修正子法、ホイン法を採用した。

理論的に安定性を確認する方法について述べる。差分式が $u_n = a_1 u_{n-1} + a_0 ...(*)$ で書けるとする。数値計算手法が不安定である時、$\lim_{n\to\infty} |u_n| = \infty$ となるはずである。このようになる条件を探す。$(*)$ を満たす数列を、$u_n = \lambda^n$ の形で探す。$u_n = \lambda^n$ を代入すると、式 $(*)$ は $\lambda^n - a_1\lambda^{n-1} - a_0 = 0$ という形の特性方程式となる。次にこの特性方程式の同次方程式 $\lambda^n - a_1\lambda^{n-1} = 0$ を解くと、$\lambda = a_1$ となる。一方で、$(*)$ の非同次解として $u_n = \frac{a_0}{1-a_1}$ が得られる。以上より、$(*)$ の解は c_1 を未定係数として $u_n = c_1 a_1^n + \frac{a_0}{1-a_1}$ となる。u_0 が与えられている条件下においては、$c_1 = u_0 - \frac{a_0}{1-a_1}$ と決定できる。これより、$(*)$ の解は $u_n = (u_0 - \frac{a_0}{1-a_1})a_1^n + \frac{a_0}{1-a_1}$ である。この解は $|a_1| > 1$ の

時 $n \to \infty$ で発散する。
数値的に安定性を確認する方法について述べる。このためには、何種類かの時刻のステップ幅を用いて時刻 t を離散化し、それぞれの時刻における u の値を各ステップ幅に対してグラフにまとめ、t の増加に従い u が収束するか否かを視覚的に判断すれば良い。今回、全ての手法において Δt の値は 0.01,0.19,0.205 とし、$t \in [0, 10]$ とした。これだけの範囲を取れば、解の安定性を確かめるのに十分なステップ数があると判断した。
まず、クランクニコルソン法を用いた解析について議論する。
クランクニコルソン法は $u_n = u_{n-1} + \frac{\Delta t}{2}(f_{n-1} + f_n)$ で表される。この式の f_n に $-10u_n + 1$ を代入し、整理すると $u_n = \frac{1-5\Delta t}{1+5\Delta t} u_{n-1} + \Delta t$ となる。この式において、式 $(*)$ の a_1 にあたるのは $\frac{1-5\Delta t}{1+5\Delta t}$ なので、解が安定となるための条件は $|\frac{1-5\Delta t}{1+5\Delta t}| < 1$ である。$|1 + 5\Delta t| > |1 - 5|\Delta t|$ は $\Delta t > 0$ の範囲において常に成り立つため、この手法は Δt の値に関わらず常に安定である。
これを数値的に確認する。結果のグラフは以下である。

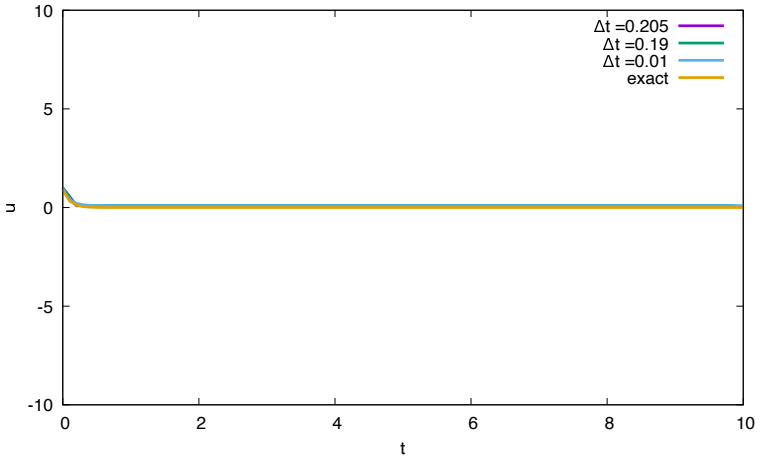

図 1: クランクニコルソン法を用いた際の u の様子

確かに、全ての Δt において安定な解が得られることが確認できた。なお、解析解である $0.9\exp(-10t) + 0.1$ を exact としてプロットしている。これと比較すると、全ての Δt において数値解が解析解と近い値となっていることがわかる。コードは以下である。

コード 6: クランクニコルソン法を用いたコード

```
1    #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  int main(){
6      //初期値など
```

```
7       double dt = 0.205;
8       double t0 = 0;
9       double tn = 10;
10      double u = 1;
11      int n = (int)tn/dt;
12      double t = t0;
13      double alpha = 10;
14      double beta = 1;
15
16      FILE *fp;//ファイルに値を出力
17      fp = fopen("kadai4_krank_0205.txt", "w");
18      fprintf(fp, "%.8f␣%.8f␣\n", t, u);
19
20      for(int i=1; i<=n; i++){
21          u = (beta * dt + u * (1−alpha*dt/2)) / (1 + alpha * dt/2);//クランクニクソン法の線形の漸化式を解いた形
22          t += dt;
23          fprintf(fp, "%.8f␣%.8f␣\n", t, u);
24      }
25
26      fclose(fp);
27  }
```

以下コードの説明を行う。変数 dt は時刻の刻み幅を表し、n はステップ数を表す。離散化して計算を繰り返し行う前に、ファイルに初期値を出力しておく。for 文内での繰り返し演算においては、クランクニクソン法の式 $u_n = u_{n-1} + \frac{\Delta t}{2}(f_{n-1} + f_n)$ に $f_n = -\alpha u_n + \beta$ を代入して得られる式である $u = \frac{\beta\Delta t + u(1-\alpha\Delta t/2)}{1+\alpha\Delta t/2}$ を用いて u の更新を行った。

次に、ホイン法を用いた解析について議論する。

ホイン法は $u_n^* = u_{n-1} + f_{n-1}\Delta t$ として、$u_n = u_{n-1} + \frac{\Delta t}{2}(f_{n-1} + f(t_n, u_n^*))$ で表される。この式の f_n に $-10u_n + 1$ を代入し、整理すると $u_n = (1 - 10\Delta t + 50(\Delta t)^2)u_{n-1} - 5(\Delta t)^2 + \Delta t$ となる。この式において、式 $(*)$ の a_1 にあたるのは $1 - 10\Delta t + 50(\Delta t)^2$ なので、解が安定となるための条件は $|1 - 10\Delta t + 50(\Delta t)^2| < 1$ である。これを解くと、$0 < \Delta t < 0.2$ であるため、この範囲内の Δt であれば解は安定となり、そうでなければ不安定となることが予想される。

これを数値的に確認する。結果のグラフは以下である。

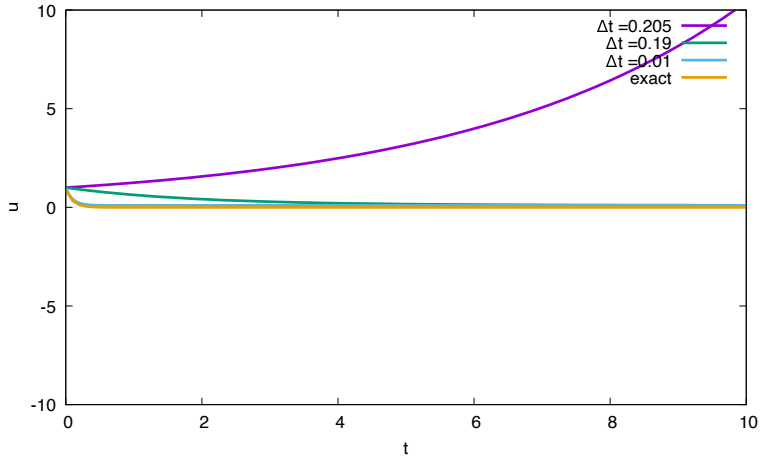

図 2: ホイン法を用いた際の u の様子

確かに、$\Delta t = 0.01, 0.19$ では安定で、$\Delta t = 0.205$ の時は発散することが確認できた。コードは以下である。

```
[captionホイン法を用いたコード=,label=code:1]
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double dudt( double u){
    double alpha = 10;
    double beta = 1;
    return −alpha ∗ u + beta;
}
//実行
int main(){
    int i;
    double u, t, k1;
    double u0 = 1;
    double t0=0;
    double tn = 10;

    u = u0;
    t = t0;
    double dt = 0.01;//変更する刻み幅..
    double n = (int)tn/dt;//ステップ数

    FILE ∗fp;//ファイルに値を出力
    fp = fopen("kadai4_hoin_001.txt", "w");
    fprintf(fp, "%.8f␣%.8f␣\n", t, u);

    // ホイン法で漸化式を計算
    for ( i=1; i <= n ; i++){
        k1 = dudt(u);
        double u_astalisk = u + k1 ∗ dt;
        u += (k1 + dudt(u_astalisk))∗dt/2;
        t += dt;
        fprintf(fp, "%.8f␣%.8f␣\n", t, u);
    }
```

```
36        fclose(fp);
37
38  }
```

このコードは、クランクニコルソン法における解析で用いたコードと同様の構造をしている。関数 dudt は u の導関数 f を表す。for 文内では、ホイン法の更新式に基づき、ステップ数の分だけ u の更新を行っている。
最後に、予測子修正子法を用いた解析について議論する。3 次の予測子修正子法では、まずアダムスバッシュ法を用いて $u_n = u_{n-1} + \frac{\Delta t}{12}(23f_{n-1} - 16f_{n-2} + 5f_{n-3})$ とする。この u_n を、アダムスムルトン法の更新式 $u_n = u_{n-1} + \frac{\Delta t}{12}(5f_n + 8f_{n-1} - f_{n-2})$ の右辺で f_n に含まれる u_n の代わりに用い、修正子として u_n を求める。2 次の予測子修正子法では、まずアダムスバッシュ法を用いて $u_n = u_{n-1} + \frac{\Delta t}{2}(3f_{n-1} - f_{n-2})$ とする。この u_n を、アダムスムルトン法の更新式 $u_n = u_{n-1} + \frac{\Delta t}{2}(f_n + f_{n-1})$ の右辺で f_n に含まれる u_n の代わりに用い、修正子として u_n を求める。
これを数値的に確認する。以下が結果のグラフである。

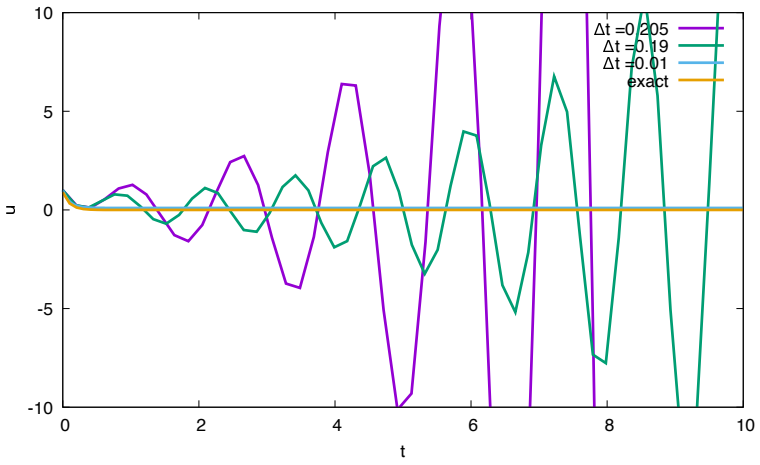

図 3: 3 次の予測子修正子法を用いた際の u の様子

この時、$\Delta t = 0.01$ では安定で、$\Delta t = 0.19, 0.205$ の時は発散することが確認できた。コードは以下である。

コード 7: 3 次の予測子修正子法を用いたコード

```
1   #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  //次のアダムスバッシュと次のアダムスムルトンで構成する 33.
6  int main(){
7       double dt = 0.205;//時間の刻み幅変更する..
8       double t0 = 0;
```

```
 9      double tn = 10;
10      double u0 = 1;
11      double u = u0;
12      int n = (int)tn/dt;//ステップ数
13      double t = t0;
14      double alpha = 10;
15      double beta = 1;
16      double f1, f2 = 0, f3 = 0;
17
18      FILE *fp;//ファイルに値を出力
19      fp = fopen("kadai4_yosokusi_0205.txt", "w");
20      fprintf(fp, "%.8f␣%.8f␣\n", t, u);
21
22      for(int i=1; i<=n; i++){//繰り返す
23          f1 = -alpha * u + beta;
24          t += dt;
25          if(i < 3){
26              u = (u0-beta/alpha) * exp(-alpha*t) + beta/alpha;
27          }
28          else{
29              double u_tilda = u + (23*f1 - 16*f2 + 5*f3) * dt/12;
30              u += (5*(-alpha*u_tilda+beta) + 8*f1 - f2)*dt/12;
31          }
32          f3 = f2; f2 = f1;
33          fprintf(fp, "%.8f␣%.8f␣\n", t, u);
34      }
35
36      fclose(fp);
37  }
```

コードの説明を行う。コードの構造は先の 2 つの手法と同じである。u の更新部分においては、u を求めるための初期値が十分でない 1 回目と 2 回目の更新では解析解である $u = (u_0 - \beta/\alpha)\exp(-\alpha t) + \beta/\alpha$、つまり $u = 0.9\exp(-10t) + 0.1$ を用いている。更新ごとに、$f_{n-1}, f_{n-2}, f_{n-3}$ を表す変数である f3,f2,f1 に対して、f2 を f3 に、f1 を f2 に代入する作業を行っている。

以下が 2 次の予測子修正子法を用いた結果である。この時も、3 次の時と同様に、$\Delta t = 0.01$ では解析解と同様の結果が得られ、$\Delta t = 0.19, 0.205$ では発散することがわかった。

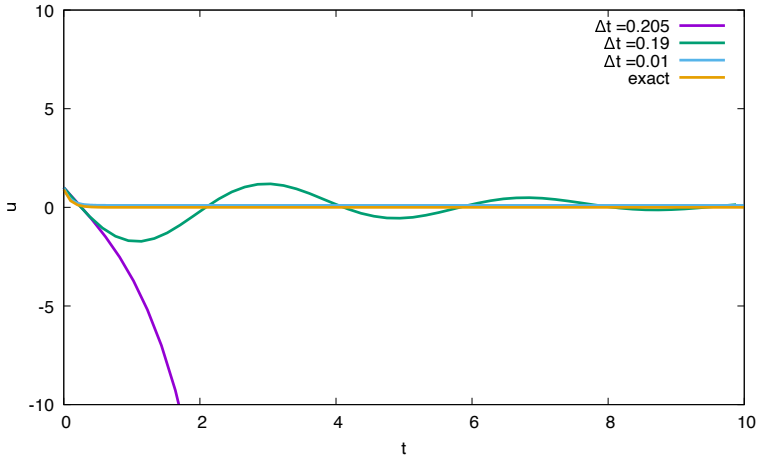

図 4: 2 次の予測子修正子法を用いた際の u の様子

以下が図の出力に用いたコードである。

コード 8: 2 次の予測子修正子法を用いたコード

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

//次のアダムスバッシュと次のアダムスムルトンで構成する 22.
int main(){
    double dt = 0.01;//時間の刻み幅変更する..
    double t0 = 0;
    double tn = 10;
    double u0 = 1;
    double u = u0;
    int n = (int)tn/dt;//ステップ数
    double t = t0;
    double alpha = 10;
    double beta = 1;
    double f1, f2 = 0;

    FILE *fp;//ファイルに値を出力
    fp = fopen("kadai4_2ji_2ji_001.txt", "w");
    fprintf(fp, "%.8f␣%.8f␣\n", t, u);

    for(int i=1; i<=n; i++){
        f1 = -alpha * u + beta;
        t += dt;
        if(i < 2){
            u = (u0-beta/alpha) * exp(-alpha*t) + beta/alpha;
        }
        else{
            double u_tilda = u + (3*f1 - f2 ) * dt/2;
            u += ((-alpha*u_tilda+beta) + f1)*dt/2;
        }
        f2 = f1;
        fprintf(fp, "%.8f␣%.8f␣\n", t, u);
    }
```

```
35
36      fclose(fp);
37  }
```

コードの構成は3次の時と同じであり、u の更新の際に $u_n = u_{n-1} + \frac{\Delta t}{2}(3f_{n-1} - f_{n-2})$ とし、この u_n を、アダムスムルトン法の更新式 $u_n = u_{n-1} + \frac{\Delta t}{2}(f_n + f_{n-1})$ の右辺で f_n に含まれる u_n の代わりに用い、修正子として u_n を求めている点が相違点である。

6 課題5

ここでは、新たに作られた数値計算手法である

$$u_n = u_{n-2} + 2f(t_{n-1}, u_{n-1})\Delta t \tag{3}$$

がある初期値問題

$$u' = -2u + 1, u(0) = 1 \tag{4}$$

を解くことができないことを示し、実際に数値計算で確認する。
まず、理論的に示す。この条件下において、u の更新の式は $u_n = u_{n-2} + 2(-2u_{n-1}+1)\Delta t$、つまり $u_n = u_{n-2} - 4\Delta t u_{n-1} + 2\Delta t$ となる。ここで、$2\Delta t$ を無視し、u_n に λ^n を代入すると、$\lambda^n - \lambda^{n-2} + 4\Delta t\lambda^{n-1} = 0$ となり、両辺を λ^{n-2} で割ると特性方程式 $\lambda^2 + 4\Delta t\lambda - 1 = 0$ が得られる。これを解くと、$\lambda = -2\Delta t \pm \sqrt{4(\Delta t)^2 + 1}$ が得られる。$\lambda = -2\Delta t - \sqrt{4(\Delta t)^2 + 1} < -1$ が常に成り立つことを示す。この式は、$-2\Delta t + 1 < \sqrt{4(\Delta t)^2 + 1}$ と同じである。$\Delta t \geq 0.5$ の時、左辺は0以下となり右辺は0より大きくなるため、不等式は常に成立する。$\Delta t \leq 0.5$ の時、両辺は常に正なので、両辺を二乗しても不等式の意味は変わらない。よって不等式は $4(\Delta t)^2 - 4\Delta t + 1 < 4(\Delta t)^2 + 1$ となる。移項すると $-4\Delta t < 0$ となる。これは $\Delta t > 0$ において常に成り立つ。以上より、特性方程式の解の一つである $\lambda = -2\Delta t - \sqrt{4(\Delta t)^2 + 1}$ について、$\lambda = -2\Delta t - \sqrt{4(\Delta t)^2 + 1} < -1$、つまり $|\lambda| = |-2\Delta t - \sqrt{4(\Delta t)^2 + 1}| > 1$ であることが示された。よってこの手法によってこの常微分方程式を解くことはできないと結論づけられる。
次に、これを数値的に確認する。$\Delta t = 0.1, 0.01, 0.001$ の時について検証を行った。それぞれ $t = 100$ になるまで更新を行った。結果は以下である。これらの結果より、確かに t が増加すると解が発散すること、そして Δt の値が小さくなるほど解が発散する時刻 $\tilde{t}$ が遅くなることが確認できた。

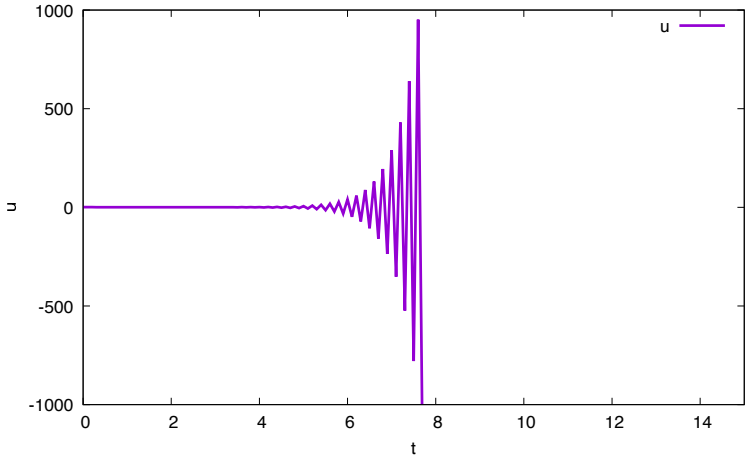

図 5: $\Delta t = 0.1$ の時の u の様子

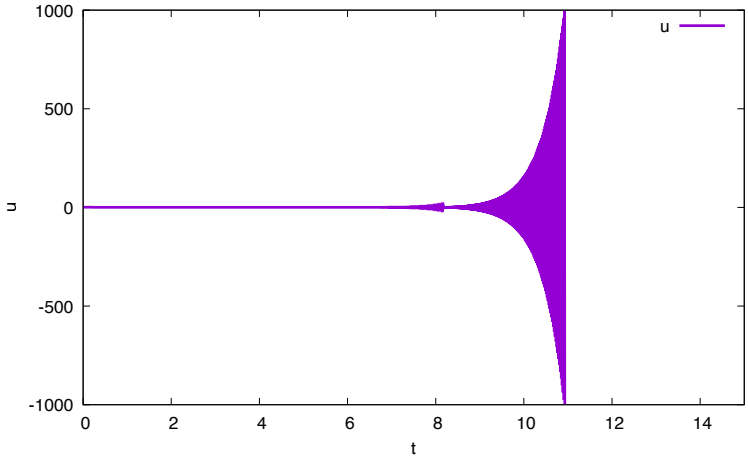

図 6: $\Delta t = 0.01$ の時の u の様子

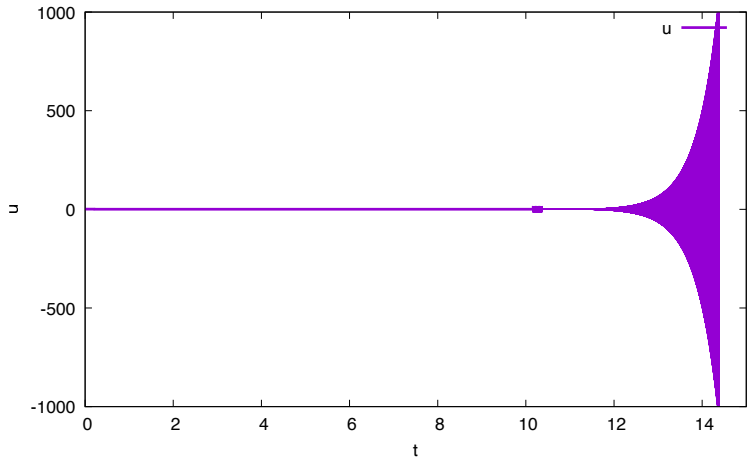

図 7: $\Delta t = 0.001$ の時の u の様子

以下がコードである。

コード 9: 課題 5 の方法で常微分方程式を解くコード

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

double dudt(double u){
    return -2 * u + 1;
}

int main(){
    double u2 = 0;
    double u1 = 1;
    double u;
    double dt = 0.1;//刻み幅.
    double t = 0;

    FILE *fp;//ファイルに値を出力
    fp = fopen("kadai5.txt", "w");
    fprintf(fp, "%.8f␣%.8f␣\n", t, u1);

    for(int i=1; i<=1000; i++){
        t += dt;
        if(i < 2){
            u = (exp(-2*t) + 1)/2;//初期値が足りない間は解析解を使用
        }
        else{
            u = u2 + 2 * dudt(u1) * dt;
        }
        fprintf(fp, "%.8f␣%.8f␣\n", t, u);
        u2 = u1; u1 = u;
    }
    fclose(fp);
}
```

コードの説明を行う。関数 dudt 内では、与えられた常微分方程式 $u' = -2u+1$

に従い、微分係数を計算する。実行部分では、刻み幅 dt および繰り返し回数を適宜変更する。掲載しているコードは $\Delta t = 0.1$、繰り返し回数 1000 の時のものである。for 文内では初期値が足りないはじめの 1 回はは解析解である $u = \frac{\exp(-2t)+1}{2}$ を使用し、その後は与えられた更新手法に基づいて更新を行った。一度更新するごとに、u_{n-1} の値を u_{n-2} に、u_n の値を u_{n-1} に代入した。

7　課題 6

ここでは、生物の個体数の増減に関する数理モデル $u' = (u-1)u$ を、u の初期値 u_0 を $0.5, 1.0, 1.5$ として数値的に解き、グラフにまとめた。また、変数変換 $\frac{du}{ds} = \frac{(u-1)u}{\sqrt{1+u'^2}}, \frac{dt}{ds} = \frac{1}{\sqrt{1+u'^2}} (u(0) = u_0, t(0) = 0)$ を用いた場合と用いない場合それぞれについての結果を比較した。手法としては、前進オイラー法を使用した。以下が結果のグラフである。$u_0 = 1.5$ のグラフにおいて、橙色の点線は $t = \log 3$ を表す。これは与えられた常微分方程式の解析解が発散する t の値である。この結果、$u_0 = 0.5$ の時は単調減少となり、$u_0 = 1.0$ の時は 1.0 の定常値となり、$u_0 = 1.5$ の時は短調増加となることがわかった。また、$u_0 = 1.5$ の時は、変数変換を行った場合、行わなかった時より解析解に近い挙動をとることがわかった。さらに、時刻の刻み幅が小さい方が、大きい方より解析解に近い挙動をとることがわかった。

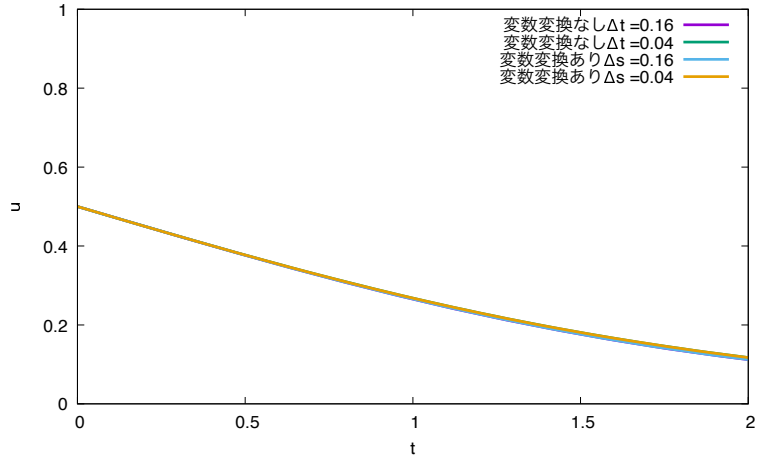

図 8: $u_0 = 0.5$ の時の u の様子

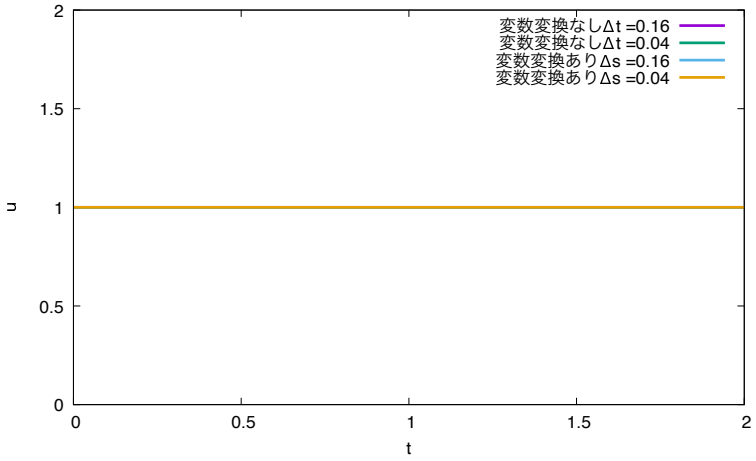

図 9: $u_0 = 1.0$ の時の u の様子

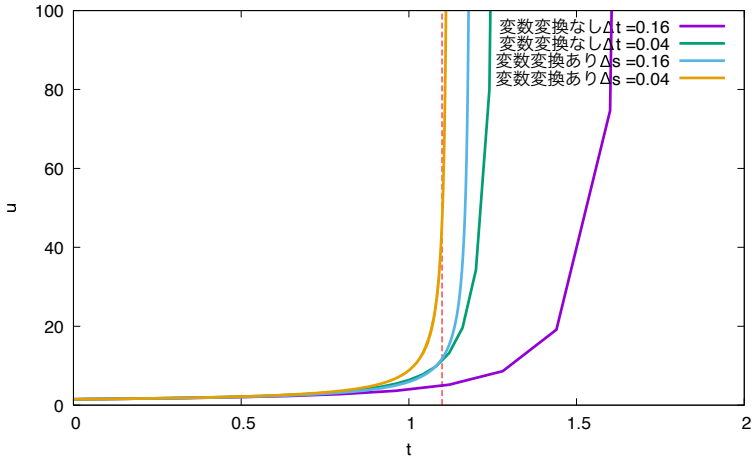

図 10: $u_0 = 1.5$ の時の u の様子

以下が変数変換せず、前進オイラー法を用いて課題 6 を解くソースコードである。

コード 10: 変数変換せずに課題 6 を解くコード

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 double dudt(double u){
6     return (u-1) * u;
7 }
```

```
int main(){
    //初期値など
    double dt = 0.04;//刻み幅変更する..
    double t0 = 0;
    double tn = 100.0;
    double u = 0.5;//u の初期値変更する..
    int n = (int)tn/dt;
    double t = t0;
    FILE *fp;//ファイルに値を出力
    fp = fopen("kadai6_nasi_05_004.txt", "w");
    fprintf(fp, "%.8f␣%.8f␣\n", t, u);

    for(int i=1; i<=n; i++){
        //変数変換せず前進オイラー法で求める.
        t += dt;
        u += dt * dudt(u);
        fprintf(fp, "%.8f␣%.8f␣\n", t, u);
    }

    fclose(fp);
}
```

以下、コードの説明を行う。
時間の刻み幅 dt は、0.04 または 0.16 に手動で変更する。掲載したコードでは 0.04 としている。また、u の初期値 u_0 は 0.5 または 1.0 または 1.5 に変更する。掲載したコードでは 0.5 としている。実行部では前進オイラー法を用い、時刻が 100 になるまで更新を行う。時刻ごとに、得られた u の値とその時刻をファイルに出力する。
以下が変数変換 $\frac{du}{ds} = \frac{(u-1)u}{\sqrt{1+u'^2}}, \frac{dt}{ds} = \frac{1}{\sqrt{1+u'^2}}(u(0) = u_0, t(0) = 0)$ を用いた前進オイラー法で課題 6 を解くコードである。

コード 11: 変数変換して課題 6 を解くコード

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

double dudt(double u){//与えられた微分方程式
    return (u−1) * u;
}

int main(){
    //初期値など
    double u0 = 0.5;//u の初期値変更する.
    double u = u0;
    double t = 0;
    double sn = 100.0;
    double ds = 0.16; //または 0.04
    double s = 0;
    int n = (int)sn/ds;

    FILE *fp;//ファイルに値を出力
    fp = fopen("kadai6_henkan_05_016.txt", "w");
    fprintf(fp, "%.8f␣%.8f␣\n", t, u);
    //実行
    for(int i=1; i<=n; i++){
        s += ds;
        //変数変換
```

```
26          double u_kari = u + ds * (u-1)*u/sqrt(1+pow((u-1),2)*pow(u
                ,2));
27          t += ds * 1/sqrt(1+pow(u,2)*pow(u-1,2));
28          fprintf(fp, "%.8f␣%.8f␣\n", t, u_kari);
29          u = u_kari;
30      }
31
32      fclose(fp);
33  }
```

以下、コードの説明を行う。
時間の刻み幅 dt は、0.04 または 0.16 に手動で変更する。掲載したコードでは 0.16 としている。また、u の初期値 u_0 は 0.5 または 1.0 または 1.5 に変更する。掲載したコードでは 0.5 としている。実行部では変数変換 $\frac{du}{ds} = \frac{(u-1)u}{\sqrt{1+u'^2}}$, $\frac{dt}{ds} = \frac{1}{\sqrt{1+u'^2}}(u(0) = u_0, t(0) = 0)$ を適用した前進オイラー法を用い、時刻が 100 になるまで更新を行う。時刻ごとに、得られた u の値とその時刻をファイルに出力する。

8 課題 7

ここでは、前進オイラー法とルンゲクッタ法を用いて、ローレンツ方程式

$$x' = 10(y - x), x(0) = 1 + \epsilon \tag{5}$$

$$y' = 28x - y - xz, y(0) = 0 \tag{6}$$

$$z' = xy - \frac{8}{3}z, z(0) = 0 \tag{7}$$

を数値的に解く。1 番では (x(t),y(t),z(t)) の軌道を x-y-z 空間上に図示し、x(t) を t の関数として図示する。2 番では $\Delta t = 0.01 * 2^{-i}(i = 0, ..., 13)$ として方程式の数値解を求め、t を固定した時の Δt の大きさによる x(t) の相違を調べる。また、数値計算の誤差が x(t) に与える影響についても考察する。3 番では $\epsilon = 0, 0.1, 0.01, 0.001$ とし、ϵ が x(t) に与える影響を考察する。
1. ここでは、ローレンツ方程式において $\epsilon = 0$ とし、前進オイラー法およびルンゲクッタ法を用いて x(t) を t の関数として図示する。また、(x(t),y(t),z(t)) の軌道を x-y-z 空間上に図示し、長時間挙動を確認する。なお、時間の刻み幅は 0.01、初期値は $t = 0, x = 1.0, y = 0, z = 0$ とし、t=100 になるまで更新を繰り返した。
結果、前進オイラー法を用いた時の (x,y,z) の挙動は以下のようになった。

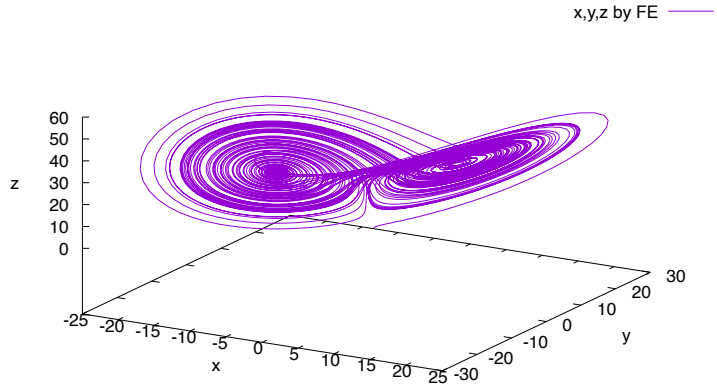

図 11: 前進オイラー法を用いた時の x,y,z の挙動

ルンゲクッタ法を用いた時の挙動は以下のようになった。

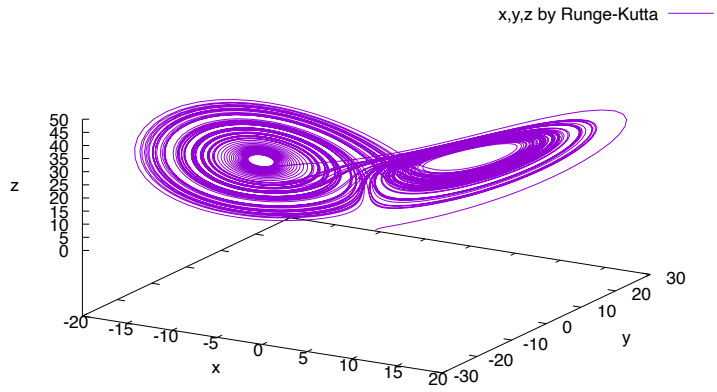

図 12: ルンゲクッタ法を用いた時の x,y,z の挙動

どちらの手法を用いても、渦巻が広がっていくような挙動を示していることから、ローレンツ方程式にのっとる x,y,z の挙動は非周期的であることが確認できた。以下が前進オイラー法を用いてローレンツ方程式を解くコードである。

コード 12: 前進オイラー法を用いてローレンツ方程式を解くコード

```
1  #include <stdio.h>
2  #include <stdlib.h>
```

```
3  #include <math.h>
4  #include <stdbool.h>
5  //ローレンツ方程式
6  double dxdt(double x, double y, double z){
7      return 10 * (y - x);
8  }
9  double dydt(double y, double x, double z){
10     return 28*x - y - x*z;
11 }
12 double dzdt(double z, double x, double y){
13     return x*y - 8*z/3;
14 }
15 //実行
16 int main(){
17     double limit = 100;
18     double f1,f2,f3,f4;
19     double x=1,y=0,z=0,x_kari,y_kari,z_kari;//初期条件
20     double dt = 0.01;
21     int n = (int)limit/dt;//繰り返し回数
22
23     FILE *fp_0;//ファイルに値を出力
24     fp_0 = fopen("kadai7_1_euler.txt", "w");
25     fprintf(fp_0, "%.8f␣%.8f␣%.8f\n", x,y,z);
26
27     for(int i=1; i<=n; i++){//更新
28         x_kari = x + dt*dxdt(x ,y ,z);
29         y_kari = y + dt*dydt(y, x, z);
30         z_kari = z + dt*dzdt(z,x,y);
31         x = x_kari;
32         y = y_kari;
33         z = z_kari;
34         fprintf(fp_0, "%.8f␣%.8f␣%.8f\n", x,y,z);
35     }
36
37     fclose(fp_0);
38
39 }
```

コードの説明を行う。dxdt,dydt,dzdt はそれぞれ x 方向、y 方向、z 方向の微分係数を表している。実行部では、x,y,z の初期値を定義し、それをファイルに出力する。for 文内において、x,y,z に前進オイラー法のアルゴリズムを適用し、結果をそれぞれ x_kari, y_kari, z_kari に格納する。x,y,z すべての更新が終了したのち、x_kari, y_kari, z_kari を x,y,z に格納することで更新を行う。その度にファイルに x,y,z の値を出力する。これを t=100 になるまで繰り返す。以下がルンゲクッタ法を用いてローレンツ方程式を解くコードである。

コード 13: ルンゲクッタ法を用いてローレンツ方程式を解くコード

```
1   #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <stdbool.h>
5
6  //ローレンツ方程式
7  double dxdt(double x, double y, double z){
8      return 10 * (y - x);
9  }
10 double dydt(double x, double y, double z){
11     return 28*x - y - x*z;
```

```
12 }
13 double dzdt(double x, double y, double z){
14     return x*y − 8*z/3;
15 }
16 //実行
17 int main(){
18     double limit = 100;
19     double epsilon=0;
20     double f1_x,f2_x,f3_x,f4_x,f1_y,f2_y,f3_y,f4_y,f1_z,f2_z,f3_z,f4_z;
21     double x=1,y=0,z=0,x_kari,y_kari,z_kari,t=0;
22     double dt = 0.01;
23     int n = (int)limit/dt;
24
25     FILE *fp_0;//ファイルに値を出力
26     fp_0 = fopen("kadai7_1_runge_xt.txt", "w");
27     fprintf(fp_0, "%.8f␣%.8f␣\n", t,x);
28     //ルンゲクッタ法
29     for(int i=1; i<=n; i++){
30             t += dt;
31
32             f1_x = dxdt(x,y,z);
33             f1_y = dydt(x,y,z);
34             f1_z = dzdt(x,y,z);
35
36             f2_x = dxdt(x + f1_x*dt/2,y + f1_y*dt/2,z + f1_z*dt/2);
37             f2_y = dydt(x + f1_x*dt/2,y + f1_y*dt/2,z + f1_z*dt/2);
38             f2_z = dzdt(x + f1_x*dt/2,y + f1_y*dt/2,z + f1_z*dt/2);
39
40             f3_x = dxdt(x + f2_x*dt/2,y + f2_y*dt/2,z + f2_z*dt/2);
41             f3_y = dydt(x + f2_x*dt/2,y + f2_y*dt/2,z + f2_z*dt/2);
42             f3_z = dzdt(x + f2_x*dt/2,y + f2_y*dt/2,z + f2_z*dt/2);
43
44             f4_x = dxdt(x + f3_x*dt,y + f3_y*dt,z + f3_z*dt);
45             f4_y = dydt(x + f3_x*dt,y + f3_y*dt,z + f3_z*dt);
46             f4_z = dzdt(x + f3_x*dt,y + f3_y*dt,z + f3_z*dt);
47
48             x_kari = x + (f1_x + 2*f2_x + 2*f3_x + f4_x)*dt/6;
49             y_kari = y + (f1_y + 2*f2_y + 2*f3_y + f4_y)*dt/6;
50             z_kari = z + (f1_z + 2*f2_z + 2*f3_z + f4_z)*dt/6;
51
52             x = x_kari;
53             y = y_kari;
54             z = z_kari;
55             fprintf(fp_0, "%.8f␣%.8f␣\n", t,x);
56     }
57     fclose(fp_0);
58 }
```

コードの説明を行う。コードの構造は前進オイラー法のコードと同じである。for 文内では、f1,f2,f3,f4 を仮の変数として用い、x,y,z それぞれに対してルンゲクッタ法を適用している。
また、前進オイラー法とルンゲクッタ法それぞれに対する x(t) の挙動は以下のようになった。

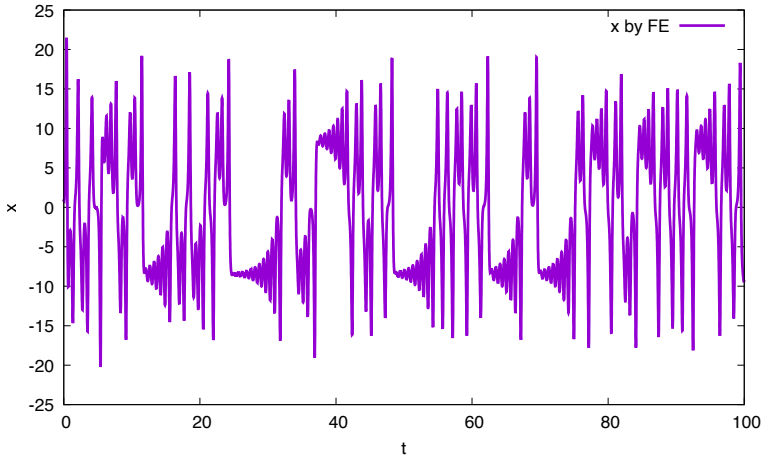

図 13: 前進オイラー法を用いた時の x の挙動

ルンゲクッタ法を用いた時の挙動は以下のようになった。

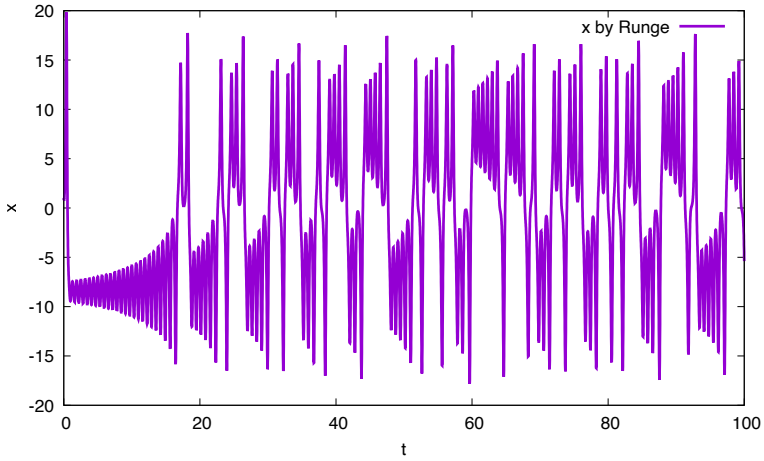

図 14: ルンゲクッタ法を用いた時の x の挙動

以上の図より、x(t) の非周期的な挙動が確認できた。以下オイラー法を用いて x(t) の解を求めるのに使用したコードである。

コード 14: オイラー法を用いて x(t) の解を求めるコード

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <stdbool.h>
5
```

```
6  double dxdt(double x, double y, double z){
7      return 10 * (y - x);
8  }
9  double dydt(double y, double x, double z){
10     return 28*x - y - x*z;
11 }
12 double dzdt(double z, double x, double y){
13     return x*y - 8*z/3;
14 }
15
16 int main(){
17     double limit = 100;
18     double epsilon=0;
19     double f1,f2,f3,f4;
20     double x=1,y=0,z=0,x_kari,y_kari,z_kari;
21     double dt = 0.01, t=0;
22     int n = (int)limit/dt;
23
24     FILE *fp_0;//ファイルに値を出力
25     fp_0 = fopen("kadai7_1_euler_xt.txt", "w");
26     fprintf(fp_0, "%.8f␣%.8f␣%.8f␣\n", x,y,z);
27
28     for(int i=1; i<=n; i++){
29         t += dt;
30         x_kari = x + dt*dxdt(x ,y ,z);
31         y_kari = y + dt*dydt(y, x, z);
32         z_kari = z + dt*dzdt(z,x,y);
33         x = x_kari;
34         y = y_kari;
35         z = z_kari;
36         fprintf(fp_0, "%.8f␣%.8f␣%.8f␣\n", x,y,z);
37     }
38
39     fclose(fp_0);
40
41 }
```

以下がルンゲクッタ法を用いて x(t) の解を求めるコードである。

コード 15: ルンゲクッタ法を用いて x(t) の解を求めるコード

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <stdbool.h>
5  //ローレンツ方程式
6  double dxdt(double x, double y, double z){
7      return 10 * (y - x);
8  }
9  double dydt(double x, double y, double z){
10     return 28*x - y - x*z;
11 }
12 double dzdt(double x, double y, double z){
13     return x*y - 8*z/3;
14 }
15 //実行
16 int main(){
17     double limit = 100;
18     double epsilon=0;
19     double f1_x,f2_x,f3_x,f4_x,f1_y,f2_y,f3_y,f4_y,f1_z,f2_z,f3_z,f4_z;
20     double x=1,y=0,z=0,x_kari,y_kari,z_kari,t=0;
21     double dt = 0.01;
22     int n = (int)limit/dt;
```

```
23
24      FILE *fp_0;//ファイルに値を出力
25      fp_0 = fopen("kadai7_1_runge_xt.txt", "w");
26      fprintf(fp_0, "%.8f␣%.8f␣\n", t,x);
27      //ルンゲクッタ法
28      for(int i=1; i<=n; i++){
29              t += dt;
30
31              f1_x = dxdt(x,y,z);
32              f1_y = dydt(x,y,z);
33              f1_z = dzdt(x,y,z);
34
35              f2_x = dxdt(x + f1_x*dt/2,y + f1_y*dt/2,z + f1_z*dt/2);
36              f2_y = dydt(x + f1_x*dt/2,y + f1_y*dt/2,z + f1_z*dt/2);
37              f2_z = dzdt(x + f1_x*dt/2,y + f1_y*dt/2,z + f1_z*dt/2);
38
39              f3_x = dxdt(x + f2_x*dt/2,y + f2_y*dt/2,z + f2_z*dt/2);
40              f3_y = dydt(x + f2_x*dt/2,y + f2_y*dt/2,z + f2_z*dt/2);
41              f3_z = dzdt(x + f2_x*dt/2,y + f2_y*dt/2,z + f2_z*dt/2);
42
43              f4_x = dxdt(x + f3_x*dt,y + f3_y*dt,z + f3_z*dt);
44              f4_y = dydt(x + f3_x*dt,y + f3_y*dt,z + f3_z*dt);
45              f4_z = dzdt(x + f3_x*dt,y + f3_y*dt,z + f3_z*dt);
46
47              x_kari = x + (f1_x + 2*f2_x + 2*f3_x + f4_x)*dt/6;
48              y_kari = y + (f1_y + 2*f2_y + 2*f3_y + f4_y)*dt/6;
49              z_kari = z + (f1_z + 2*f2_z + 2*f3_z + f4_z)*dt/6;
50
51              x = x_kari;
52              y = y_kari;
53              z = z_kari;
54              fprintf(fp_0, "%.8f␣%.8f␣\n", t,x);
55      }
56      fclose(fp_0);
57  }
```

x(t) を求めるのに使用したコードは、x,y,z を求めるのに使用したコードと同じ構造をしている。相違点は、ファイルに出力する値を (t,x) としている点のみである。

2. ここでは、$\Delta t = 0.01 * 2^{-i} (i = 0, ..., 13)$ として時間を離散化し、ローレンツ方程式の数値解をそれぞれの i に対して求める。具体的には、$t = 15, 30, 60$ の 3 つの時刻それぞれにおいて、横軸を i、縦軸を $x(t)$ としたグラフを作成し、数値計算の誤差が $x(t)$ に与える影響を議論する。今回、x の初期値 $1 + \epsilon$ における ϵ は 0.01 とした。

結果、$t = 15, 30, 60$ の時の挙動はそれぞれ以下のようになった。

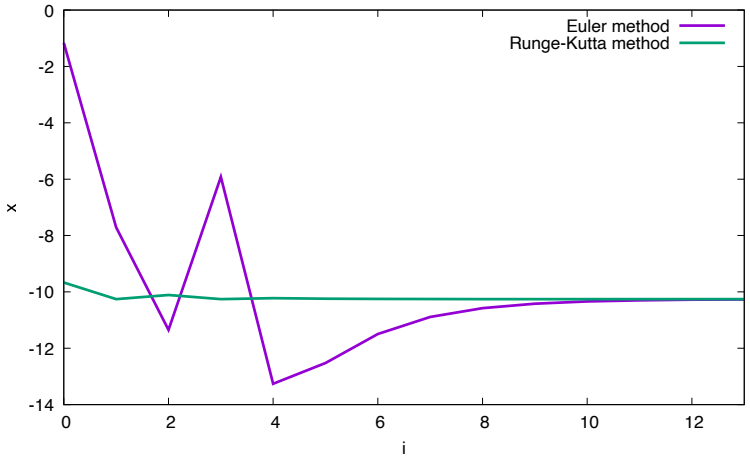

図 15: $t = 15$ の時の、前進オイラー法、ルンゲクッタ法それぞれで求めた x(t) の値. 横軸 i、縦軸 x(t).

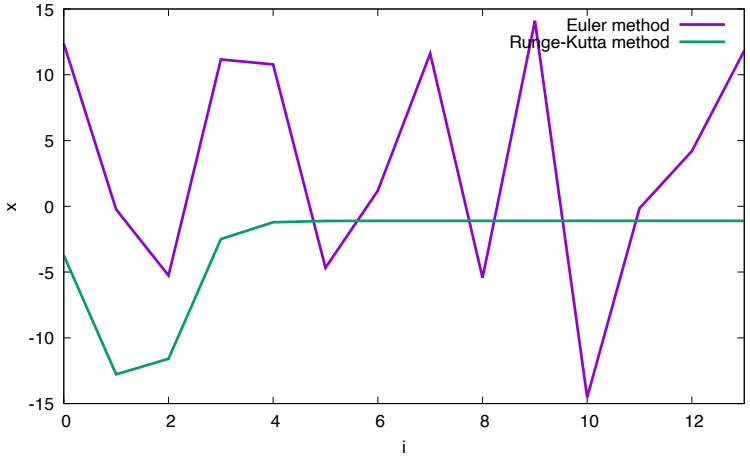

図 16: $t = 30$ の時の、前進オイラー法、ルンゲクッタ法それぞれで求めた x(t) の値. 横軸 i、縦軸 x(t).

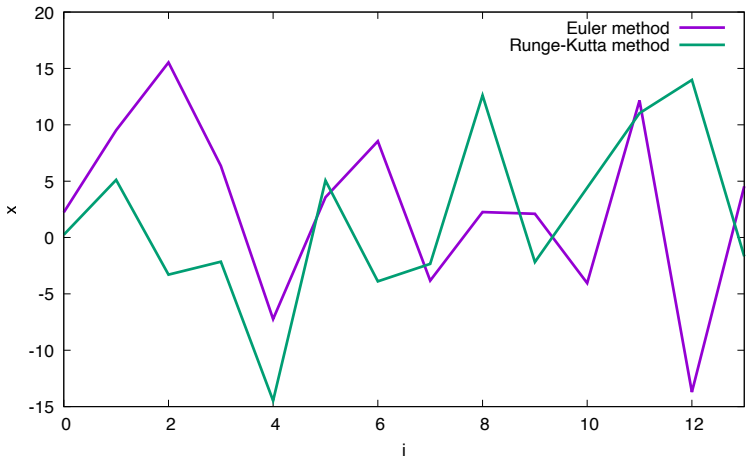

図 17: $t = 60$ の時の、前進オイラー法、ルンゲクッタ法それぞれで求めた x(t) の値. 横軸 i、縦軸 x(t).

このグラフからわかることを述べる。まず、$t = 15$ の時、オイラー法は Δt が大きい場合は数値にブレがあるが、Δt が小さくなるにつれて正確な値を求められるようになることがわかる。また、ルンゲクッタ法を用いると、大きな Δt の値でも正確な値を求められることがわかる。$t = 30$ の時、オイラー法を用いると Δt を小さくしても正確な値を求めるのが困難になることがわかる。一方ルンゲクッタ法を用いると、Δt が大きい時は解が不安定なものの、小さくなるにつれて安定した解を求められるようになることがわかる。$t = 60$ の時、オイラー法、ルンゲクッタ法ともに Δt の値にかかわらず解が不安定となることがわかる。
基本的にルンゲクッタ法の方がオイラー法より安定性が高いのは、前者の方が手法としての次数が大きく、より精密な解を求められるからだと考えられる。また、Δt が小さい方が解が正確になるのは、より多くのステップ数で数値解を求められるからだと考えられる。加えて、t の値が大きくなるほどどちらの手法でも解が不安定になるのは、時間が経つにつれて離散的な数値計算の誤差の影響が大きくなるためだと考えられる。
オイラー法を用いて $t = 15, 30, 60$ の時の $x(t)$ を求めるコードは以下である。

コード 16: オイラー法を用い t が 15 と 30 と 60 の時の x を出力するコード

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <stdbool.h>
5
6  int main(){
7      double epsilon = 0.01;
8      double limit = 100;
9
```

```
10      FILE *fp_15;//ファイルに値を出力
11      FILE *fp_30;
12      FILE *fp_60;
13      fp_15 = fopen("kadai7_euler_15.txt", "w");
14      fp_30 = fopen("kadai7_euler_30.txt", "w");
15      fp_60 = fopen("kadai7_euler_60.txt", "w");
16      //i=0~13 とした時の刻み幅において、t=15,30,60 の時の x を求める.
17      for(int i=0; i<=13; i++){
18          //初期値
19          double x = 1 + epsilon;
20          double y = 0;
21          double z = 0;
22          double t = 0;
23          double x_kari,y_kari,z_kari;
24          double dt = 0.01 * pow(2, -i);
25          int n = (int)limit/dt;
26          bool recorded_15=false, recorded_30=false, recorded_60=false;//t
                =15,30,60 の時の x を記録したかど
                うか
27
28          for(int k=1; k<=n; k++){//前進オイラー法
29              t += dt;
30              x_kari = x + (10 * (y - x) * dt);
31              y_kari = y + ((28*x - y - x*z) * dt);
32              z_kari = z + (x*y - 8*z/3)*dt;
33              x = x_kari;
34              y = y_kari;
35              z = z_kari;
36
37              //tの時の=15,30,60x の値を記録. 整数に丸めてその値になる
                    t のうち最初のものを抽出.
38              if((int)t == 15 && recorded_15==false){
39                  fprintf(fp_15, "%d␣%.8f␣\n", i, x);
40                  recorded_15=true;
41              }
42              else if ((int)t == 30 && recorded_30==false){
43                  fprintf(fp_30, "%d␣%.8f␣\n", i, x);
44                  recorded_30=true;
45              }
46              else if((int)t == 60 && recorded_60==false)
47                  {fprintf(fp_60, "%d␣%.8f␣\n", i, x);
48                  recorded_60=true;
49              }
50          }
51      }
52      fclose(fp_15);
53      fclose(fp_30);
54      fclose(fp_60);
55  }
```

コードの説明を行う。x, y, z に対して初期値を設定する。$t = 100$ になるまで前進オイラー法を x, y, z に適用する。$\Delta t = 2^{-i}$ の時、t には必ずちょうど 15,30,60 の値をとる時がある。よって、t を整数値に丸めた値が初めて 15,30,60 となる時、t は浮動小数点で表しても 15,30,60 となっていると言える。したがって、t を整数値に丸めた値が初めて 15,30,60 になった時、x の値をファイルに出力するようにする。t が 15,30,60 となる時の x の値を一度記録したあとは、$(int)t = 15, 30, 60$ となる他の t での x を記録しないようにするため、recorded という boolean 変数を設けている。以下がルンゲクッタ法を用いた

コードである。

コード 17: ルンゲクッタ法を用い t が 15 と 30 と 60 の時の x を出力するコード

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

//ローレンツ方程式
double dxdt(double x, double y, double z){
    return 10 * (y - x);
}
double dydt(double x, double y, double z){
    return 28*x - y - x*z;
}
double dzdt(double x, double y, double z){
    return x*y - 8*z/3;
}

int main(){
    double epsilon = 0.01;
    double limit = 100;

    FILE *fp_15;//ファイルに値を出力
    FILE *fp_30;
    FILE *fp_60;
    fp_15 = fopen("kadai7_runge_15.txt", "w");
    fp_30 = fopen("kadai7_runge_30.txt", "w");
    fp_60 = fopen("kadai7_runge_60.txt", "w");

    //の値を変える dt
    for(int i=0; i<=13; i++){
        //初期値など
        double x = 1 + epsilon;
        double y = 0;
        double z = 0;
        double t = 0;
        double dt = 0.01 * pow(2, -i);
        int n = (int)limit/dt;
        double f1_x,f2_x,f3_x,f4_x,f1_y,f2_y,f3_y,f4_y,f1_z,f2_z,f3_z,f4_z;//
            仮の変
            数
        double x_kari,y_kari,z_kari;
        bool recorded_15=false, recorded_30=false, recorded_60=false;//t
            =15,30,60 で記録されたかど
            うか

        //ルンゲクッタ法
        for(int k=1; k<=n; k++){

            t += dt;

            f1_x = dxdt(x,y,z);
            f1_y = dydt(x,y,z);
            f1_z = dzdt(x,y,z);

            f2_x = dxdt(x + f1_x*dt/2,y + f1_y*dt/2,z + f1_z*dt/2);
            f2_y = dydt(x + f1_x*dt/2,y + f1_y*dt/2,z + f1_z*dt/2);
            f2_z = dzdt(x + f1_x*dt/2,y + f1_y*dt/2,z + f1_z*dt/2);

            f3_x = dxdt(x + f2_x*dt/2,y + f2_y*dt/2,z + f2_z*dt/2);
```

```
55              f3_y = dydt(x + f2_x*dt/2,y + f2_y*dt/2,z + f2_z*dt/2);
56              f3_z = dzdt(x + f2_x*dt/2,y + f2_y*dt/2,z + f2_z*dt/2);
57
58              f4_x = dxdt(x + f3_x*dt,y + f3_y*dt,z + f3_z*dt);
59              f4_y = dydt(x + f3_x*dt,y + f3_y*dt,z + f3_z*dt);
60              f4_z = dzdt(x + f3_x*dt,y + f3_y*dt,z + f3_z*dt);
61
62              x_kari = x + (f1_x + 2*f2_x + 2*f3_x + f4_x)*dt/6;
63              y_kari = y + (f1_y + 2*f2_y + 2*f3_y + f4_y)*dt/6;
64              z_kari = z + (f1_z + 2*f2_z + 2*f3_z + f4_z)*dt/6;
65
66              x = x_kari;
67              y = y_kari;
68              z = z_kari;
69
70              //t=15,30,60 なら出力
71              if((int)t == 15 && recorded_15==false){fprintf(fp_15, "%d␣
                    %.8f␣\n", i, x);
72                  recorded_15=true;
73                  }
74              else if ((int)t==30 && recorded_30==false){ fprintf(fp_30,
                    "%d␣%.8f␣\n", i, x);
75                  recorded_30=true;
76                  }
77              else if((int)t==60 && recorded_60==false) {fprintf(fp_60, "
                    %d␣%.8f␣\n", i, x);
78                  recorded_60=true;
79              }
80          }
81      }
82      fclose(fp_15);
83      fclose(fp_30);
84      fclose(fp_60);
85  }
```

このコードは、前進オイラー法を用いたコードと同じ構造を持っている。更新の際にルンゲクッタ法を適用しているのが相違点である。

3. ここでは、$\epsilon = 0, 0.1, 0.01, 0.001$ とし、ルンゲクッタ法を用いて与えられたローレンツ方程式を解く。この際、$\epsilon = 0, 0.1, 0.01, 0.001$ のそれぞれに対して横軸 t、縦軸 x のグラフを出力し、ϵ が $x(t)$ に与える影響を議論する。なお、この小問では時間の刻み幅 dt を $1.0 * 10^{-5}$ とし、時刻 t が 30 となるまで x を出力した。これは t が 100 まで到達するのを待たずとも、ϵ が x に与える影響を議論するのに十分な結果が得られると判断したためである。

結果のグラフは以下である。このグラフより、およそ $t = 23$ のあたりまでは ϵ の値にかかわらず同じような x の値が得られるが、それ以降はズレが生じ始め、そのズレは時間が経てば経つほど大きくなるということが確認できた。

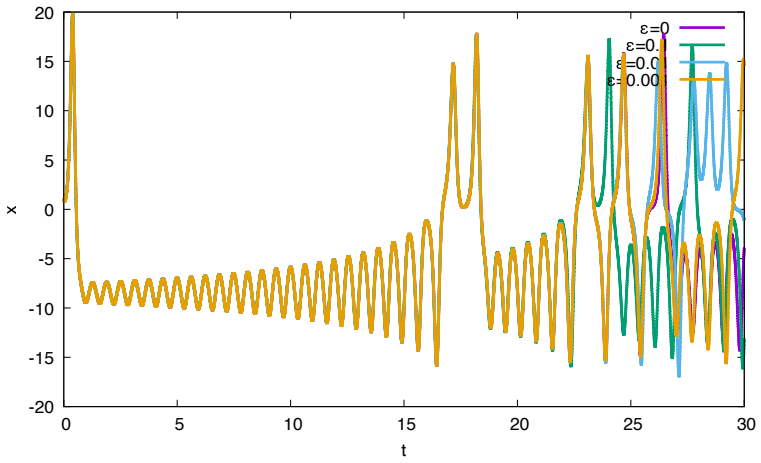

図 18: $\epsilon = 0, 0.1, 0.01, 0.001$ の時の $x(t)$ の値. 横軸 t, 縦軸 $x(t)$

以下がコードである。

コード 18: ϵ を 0 か 0.1 か 0.01 か 0.001 とし、ローレンツ方程式を解くコード

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

//ローレンツ方程式
double dxdt(double x, double y, double z){
    return 10 * (y - x);
}
double dydt(double x, double y, double z){
    return 28*x - y - x*z;
}
double dzdt(double x, double y, double z){
    return x*y - 8*z/3;
}

//実行部
int main(){
    double limit = 30;
    double epsilon;
    double f1_x,f2_x,f3_x,f4_x,f1_y,f2_y,f3_y,f4_y,f1_z,f2_z,f3_z,f4_z;
    double x_kari,y_kari,z_kari;
    double dt = pow(0.1, 5);//時間の刻み幅
    int n = (int)limit/dt;//繰り返し回数

    FILE *fp_0;//ファイルに値を出力
    FILE *fp_01;
    FILE *fp_001;
    FILE *fp_0001;
    fp_0 = fopen("kadai7_3_0.txt", "w");
    fp_01 = fopen("kadai7_3_01.txt", "w");
    fp_001 = fopen("kadai7_3_001.txt", "w");
    fp_0001 = fopen("kadai7_3_0001.txt", "w");
```

```
34
35      for(int i=0; i<=3; i++){//epsilon の値を変える
36          //初期値
37          double t = 0;
38          double x;
39          double y = 0;
40          double z = 0;
41
42          if(i == 0){
43              epsilon = 0;
44              x = 1 + epsilon;
45              fprintf(fp_0, "%.8f␣%.8f␣\n", t, x);
46          }
47          else if(i == 1){
48              epsilon = 0.1;
49              x = 1 + epsilon;
50              fprintf(fp_01, "%.8f␣%.8f␣\n", t, x);
51          }
52          else if(i == 2){
53              epsilon = 0.01;
54              x = 1 + epsilon;
55              fprintf(fp_001, "%.8f␣%.8f␣\n", t, x);
56          }
57          else if(i == 3){
58              epsilon = 0.001;
59              x = 1 + epsilon;
60              fprintf(fp_0001, "%.8f␣%.8f␣\n", t, x);
61          }
62
63          for(int k=1; k<=n; k++){//ルンゲクッタ法
64              t += dt;
65
66              f1_x = dxdt(x,y,z);
67              f1_y = dydt(x,y,z);
68              f1_z = dzdt(x,y,z);
69
70              f2_x = dxdt(x + f1_x*dt/2,y + f1_y*dt/2,z + f1_z*dt/2);
71              f2_y = dydt(x + f1_x*dt/2,y + f1_y*dt/2,z + f1_z*dt/2);
72              f2_z = dzdt(x + f1_x*dt/2,y + f1_y*dt/2,z + f1_z*dt/2);
73
74              f3_x = dxdt(x + f2_x*dt/2,y + f2_y*dt/2,z + f2_z*dt/2);
75              f3_y = dydt(x + f2_x*dt/2,y + f2_y*dt/2,z + f2_z*dt/2);
76              f3_z = dzdt(x + f2_x*dt/2,y + f2_y*dt/2,z + f2_z*dt/2);
77
78              f4_x = dxdt(x + f3_x*dt,y + f3_y*dt,z + f3_z*dt);
79              f4_y = dydt(x + f3_x*dt,y + f3_y*dt,z + f3_z*dt);
80              f4_z = dzdt(x + f3_x*dt,y + f3_y*dt,z + f3_z*dt);
81
82              x_kari = x + (f1_x + 2*f2_x + 2*f3_x + f4_x)*dt/6;
83              y_kari = y + (f1_y + 2*f2_y + 2*f3_y + f4_y)*dt/6;
84              z_kari = z + (f1_z + 2*f2_z + 2*f3_z + f4_z)*dt/6;
85
86              x = x_kari;
87              y = y_kari;
88              z = z_kari;
89
90              //epsilon の値に応じて別のファイルに出力
91              if(i == 0){fprintf(fp_0, "%.8f␣%.8f␣\n", t, x);
92                  }
93              else if (i == 1){ fprintf(fp_01, "%.8f␣%.8f␣\n", t, x);
94                  }
95              else if(i == 2){fprintf(fp_001, "%.8f␣%.8f␣\n", t, x);
96                  }
```

```
97                  else if(i == 3){fprintf(fp_0001, "%.8f␣%.8f␣\n", t, x);
98                      }
99
100             }
101         }
102         fclose(fp_0);
103         fclose(fp_01);
104         fclose(fp_001);
105         fclose(fp_0001);
106 }
```

以下コードの説明を行う。dxdt,dydt,dzdt でローレンツ方程式を定義する。時間の刻み幅および繰り返し回数 n を定義する。はじめの for 文内において、i の値が 0 ならば $\epsilon = 0$、1 ならば $\epsilon = 0.1$、2 ならば $\epsilon = 0.01$、3 ならば $\epsilon = 0.001$ と設定する。次の for 文内において、ルンゲクッタ法の更新を n 回適用する。値を更新するたびに、t と x を ϵ の値に応じたファイルに出力する。

9 まとめ

このレポートでは、全体にわたり常微分方程式の数値解法に関する問題を解き、議論を行った。課題 1 では RL 直列回路の状態方程式を定式化した。課題 2 では、4 次のアダムスバッシュホース法 $u_n = u_{n-1} + \Delta t(-\frac{9}{24}f_{n-4} + \frac{37}{24}f_{n-3} - \frac{59}{24}f_{n-2} + \frac{55}{24}f_{n-1})$ を導出し、また 4 次のアダムスムルトン法 $u_n = u_{n-1} + \Delta t(\frac{1}{24}f_{n-3} - \frac{5}{24}f_{n-2} + \frac{19}{24}f_{n-1} + \frac{9}{24}f_n)$ を導出した。さらにそれらを用いて予測子修正子法を構成した。課題 3 では具体的な常微分方程式の初期値問題 $u' = u, u(0) = 1$ を $t \in [0, 1]$ の範囲内で数値的に解き、解析解である $u = \exp(t)$ と比較した。具体的には、ステップ幅が $\Delta t = \frac{1}{2^i}$ の条件下で得られた数値解を u_n^i と表し、i の値を 1,2,...,8 の範囲で変更した計算を行った。そして、比較の際は、$t = 1$ における数値解 u_N^i と解析解の差である $E^i = |u_N^i - u(1)|$ を用い、p を計算手法の次数とし $E_r^i = \frac{E^i}{E^{i-1}}$(i=2,...,8) とすると、$E_r^i$ は i の増加とともに $\frac{1}{2^p}$ に近づくことを確認した。課題 4 では、$t \in [0, \infty]$ の範囲内において、常微分方程式の初期値問題 $u' = -10u + 1, u(0) = 1$ をクランクニコルソン法、予測子修正子法、ホイン法で理論的、数値的に解き、安定性を確認した。課題 5 では、新たに作られた数値計算手法 $u_n = u_{n-2} + 2f(t_{n-1}, u_{n-1})\Delta t$ が初期値問題 $u' = -2u + 1, u(0) = 1$ を解くことができないことを示し、実際に数値計算で確認した。課題 6 では、生物の個体数の増減に関する数理モデルの一つである $u' = (u - 1)u$ を、u の初期値 u_0 を $0.5, 1.0, 1.5$ として前進オイラー法を用いて数値的に解き、グラフにまとめた。そこでは変数変換 $\frac{du}{ds} = \frac{(u-1)u}{\sqrt{1+u'^2}}, \frac{dt}{ds} = \frac{1}{\sqrt{1+u'^2}}(u(0) = u_0, t(0) = 0)$ を用いた場合と用いない場合それぞれについての結果を比較し、結果、$u_0 = 0.5$ の時は単調減少となり、$u_0 = 1.0$ の時は 1.0 の定常値となり、$u_0 = 1.5$ の時は短調増加となることがわかった。また、$u_0 = 1.5$ の時は、変数変換を行った場合、行わなかった時より解析解に近い挙動をとることがわかった。さらに、時刻の刻み幅が小さ

い方が、大きい方より解析解に近い挙動をとることがわかった。課題 7 では、前進オイラー法とルンゲクッタ法を用いてローレンツ方程式の初期値問題を数値解析した。具体的には、x(t) を t の関数として図示し、(x(t),y(t),z(t)) の軌道を x-y-z 空間上に図示することで、ローレンツ方程式が非周期的な関数であることを確認した。また $\Delta t = 0.01 * 2^{-i} (i = 0, ..., 13)$ として方程式の数値解を求め、t を固定した時の Δt の大きさによる x(t) の相違を調べた。結果、Δt が小さい方が安定した解を求められることが確認できた。さらに、$\epsilon = 0, 0.1, 0.01, 0.001$ として x(t) を出力することで、t ががある一定の値よりも大きくなると、ϵ が x(t) に与える影響が無視できなくなることを確認した。