

シミュレーション レポート

第6回～第10回

提出日 2020年7月15日 1～2コマ目

組番号 408

学籍番号 17406

氏名 金澤雄大

1 目的

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 6

本章では課題 6 における課題内容, プログラムの説明, 実行結果, 考察の 4 つについて述べる.

3.1 課題内容

式 (1) の方程式はロトカ・ヴォルテラの方程式と呼ばれる微分方程式である. ロトカ・ヴォルテラの方程式は 2 種類の生物の個体数の変化に関する数理モデルである. 式 (1) は被食者 (食べられる側) の個体数 y_1 , 捕食者 (食べる側) の個体数 y_2 としたときの個体数の変化を表している. 本課題では式 (1) をオイラー法で実装し, 正の実数定数 a, b, c, d を変化させたときの解の特徴について考察する.

$$\begin{cases} \frac{dy_1}{dx} = ay_1 - cy_1y_2 \\ \frac{dy_2}{dx} = -by_1 + dy_1y_2 \end{cases} \quad (1)$$

3.2 プログラムの説明

本節では実装のための理論および次に示す 6 つの関数の機能, およびソースコードについて説明する.

1. multipleMatrix 関数
2. addVector 関数
3. scalerVector 関数
4. transformVector 関数
5. setVector 関数
6. main 関数

3.2.1 実装のための理論

本項では、はじめに 2 変数の単純な連立微分方程式について考え、一般変数に理論を拡張する。はじめに 2 変数の単純な連立方程式について考える。式 (2) の微分方程式をオイラー法で解く方法を考える。ただし、 a に添え字がついているものはすべて定数である。注目してほしいのは右辺で、 y_1, y_2 の一次式になっていることがわかる。

$$\begin{cases} \frac{dy_1}{dx} = a_{11} + a_{12}y_1 + a_{13}y_2 \\ \frac{dy_2}{dx} = a_{21} + a_{22}y_1 + a_{23}y_2 \end{cases} \quad (2)$$

オイラー法を用いると式 (2) は時間 $t+1$ のとき式 (3) のように近似できる。ただし、 h は刻み幅とする。

$$\begin{cases} y_1(t+1) = y_1(t) + h(a_{11} + a_{12}y_1(t) + a_{13}y_2(t)) \\ y_2(t+1) = y_2(t) + h(a_{21} + a_{22}y_1(t) + a_{23}y_2(t)) \end{cases} \quad (3)$$

式 (3) を実装すれば 2 変数の場合には問題なく実行できる。しかし、これを 3 変数、4 変数と拡張しようとすると C 言語では不都合が生じる。例えば式 (2) の右辺を関数として実装した場合、変数が増えるごとに関数の数が増えてしまい拡張性やメンテナンス性が悪い。このため式 (3) では一般変数に対応できない。そこで行列を用いる。式 (3) を行列を用いて表すと式 (4) のようになる。

$$\begin{pmatrix} y_1(t+1) \\ y_2(t+1) \end{pmatrix} = \begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix} + h \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \begin{pmatrix} 1 \\ y_1(t) \\ y_2(t) \end{pmatrix} \quad (4)$$

式 (4) を参考にして、一般に y_1, y_2, \dots, y_n の n 個の連立微分方程式 (式 5) があるとき、オイラー法による数値解は式 (6) の行列を用いることで計算できる。本課題では式 (6) を実装する。

$$\begin{cases} \frac{dy_1}{dx} = a_{11} + a_{12}y_1 + a_{13}y_2 + \dots + a_{1n+1}y_n \\ \frac{dy_2}{dx} = a_{21} + a_{22}y_1 + a_{23}y_2 + \dots + a_{2n+1}y_n \\ \vdots \\ \frac{dy_n}{dx} = a_{n1} + a_{n2}y_1 + a_{n3}y_2 + \dots + a_{nn+1}y_n \end{cases} \quad (5)$$

$$\begin{pmatrix} y_1(t+1) \\ y_2(t+1) \\ \vdots \\ y_n(t+1) \end{pmatrix} = \begin{pmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_n(t) \end{pmatrix} + h \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n+1} \\ a_{21} & a_{22} & \dots & a_{2n+1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn+1} \end{pmatrix} \begin{pmatrix} 1 \\ y_1(t) \\ y_2(t) \\ \vdots \\ y_n(t) \end{pmatrix} \quad (6)$$

式 (1) のロトカ・ヴォルテラの方程式は y_1y_2 という相互作用項を含んでいるが、式 (6) の行列によるオイラー法の数値計算はこれに対応していない。式 (1) の方程式を行列で表すと式 (7) のようになる。これによって相互作用項に対応させることができる。式 (7) から、行列の積、ベクトルの変換、ベクトルのスカラー倍、ベクトルの和の 4 つの演算が必要であることがわかる。しかし、式 (7) では $y_1(t+1) \cdot y_2(t+1) = y_1y_2(t+1)$ が成立していないため、 $y_1(t)y_2(t)$ を $y_1(t) \cdot y_2(t)$ に置き換える必要がある。

$$\begin{pmatrix} y_1(t+1) \\ y_2(t+1) \\ y_1y_2(t+1) \end{pmatrix} = \begin{pmatrix} y_1(t) \\ y_2(t) \\ y_1y_2(t) \end{pmatrix} + h \begin{pmatrix} 0 & a & 0 & -c \\ 0 & 0 & -b & d \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ y_1(t) \\ y_2(t) \\ y_1y_2(t) \end{pmatrix} \quad (7)$$

3.2.2 multipleMatrix 関数

multipleMatrix 関数は行列の積を計算する関数である. 表 2 に multipleMatrix 関数の機能, 引数, 戻り値の 3 つを示す. multipleMatrix 関数は $n \times (n + 1)$ 行列と $(n + 1) \times 1$ 行列の積を計算する機能を持つから, 引数として a,b,c の 3 つの行列をとる設計になっている. そして引数 a,b の行列積 ab を c に代入する処理を行う. ただし,DIM はオブジェクト形式マクロで定義されている微分方程式の変数の数である.

表 2: multipleMatrix 関数の機能, 引数, 戻り値

機能	$n \times (n + 1)$ 行列と $(n + 1) \times 1$ 行列の行列積を計算する
引数	double a[DIM][DIM+1],double b[DIM+1][1],double c[DIM][1]
戻り値	なし

リスト 1 に multipleMatrix 関数のソースコードを示す. リスト 1 では引数の行列 a,b の積を for 文を用いて計算し, その結果を引数 c に代入する処理を行っている.

リスト 1: multipleMatrix 関数のコード

```
1  #define DIM 3
2
3  void multipleMatrix(double a[DIM][DIM+1],double b[DIM+1][1],double c[DIM][1]){
4      int i,j,k;
5      double tmp;
6      tmp=0;
7      for(i=0; i<DIM+1; i++){
8          for(j=0; j<DIM; j++){
9              c[i][j]=0;
10             for(k=0; k<DIM+1; k++){
11                 c[i][j]+=a[i][k]*b[k][j];
12             }
13         }
14     }
15 }
```

3.2.3 addVector 関数

addVector 関数はベクトルの和を計算する関数である. 表 3 に addVector 関数の機能, 引数, 戻り値の 3 つを示す. addVector 関数は 2 つの n 次元ベクトルの和を計算する機能を持つから, 引数として a,b,c の 3 つのベクトルをとる設計になっている. そして, 引数 a,b の和を計算し,c に代入する処理を行う.

表 3: addVector 関数の機能, 引数, 戻り値

機能	2 つの n 次元ベクトルの和を計算する
引数	double a[DIM][1],double b[DIM][1],double c[DIM][1]
戻り値	なし

リスト 2 に addVector 関数のソースコードを示す. リスト 2 では引数 a,b の和を for 文を用いて計算し, その結果を引数 c に代入する処理を行っている.

リスト 2: addVector 関数のコード

```
1  void addVector(double a[DIM][1],double b[DIM][1],double c[DIM][1]){
2      int i;
3      for(i=0; i<DIM;i++){
4          c[i][0] = a[i][0]+b[i][0];
5      }
```

```

5     }
6 }

```

3.2.4 scalerVector 関数

scalerVector 関数はベクトルのスカラー倍を計算する関数である。表 4 に scalerVector 関数の機能, 引数, 戻り値の 3 つを示す。scalerVector 関数はベクトルのスカラー倍を計算する機能を持つから, 引数としてベクトル a, スカラー h をとる設計になっている。そして, 引数 a の h 倍を計算し, a を更新する処理を行う。

表 4: scalerVector 関数の機能, 引数, 戻り値

機能	ベクトルのスカラー倍を計算する
引数	double a[DIM][1], double h
戻り値	なし

リスト 3 に scalerVector 関数のソースコードを示す。リスト 3 では引数 a の h 倍を for 文を用いて計算し, その結果を a に代入する処理を行っている。

リスト 3: scalerVector 関数のコード

```

1 void scalerVector(double a[DIM][1], double h){
2     int i;
3     for(i=0; i<DIM; i++){
4         a[i][0] *= h;
5     }
6 }

```

3.2.5 transformVector 関数

transformVector 関数は n 次元ベクトル $x = (x_1, x_2, \dots, x_n)$ を $(n+1)$ 次元ベクトル $x' = (1, x_1, x_2, \dots, x_n)$ に変換する関数である。表 5 に transformVector 関数の機能, 引数, 戻り値の 3 つを示す。transformVector 関数はベクトル変換をする機能を持つから, 引数として n 次元ベクトル a および $(n+1)$ 次元ベクトル b をとる設計になっている。そして, 引数 a の変換を計算し, b に代入する処理を行う。

表 5: transformVector 関数の機能, 引数, 戻り値

機能	ベクトルの変換を行う
引数	double a[DIM][1], double b[DIM+1][1]
戻り値	なし

リスト 4 に transformVector 関数のソースコードを示す。リスト 4 では b[0][0] に 1 を代入し, 他の b 成分には引数 a の成分を for 文を用いて代入する処理を行っている。

リスト 4: transformVector 関数のコード

```

1 void transformVector(double a[DIM][1], double b[DIM+1][1]){
2     int i;
3     b[0][0]=1;
4     for(i=1; i<DIM+1; i++){
5         b[i][0] = a[i-1][0];
6     }
7 }

```

3.2.6 setVector 関数

setVector 関数はベクトルの値を別の変数に代入する関数である. 表 6 に setVector 関数の機能, 引数, 戻り値の 3 つを示す. setVector 関数はベクトルの値を別の変数に代入する機能を持つから, 引数としてベクトル a, b をとる設計になっている. そして, 引数 a を引数 b に代入する処理を行う.

表 6: setVector 関数の機能, 引数, 戻り値

機能	ベクトルの値を別の変数に代入する
引数	double a[DIM][1], double b[DIM][1]
戻り値	なし

リスト 5 に setVector 関数のソースコードを示す. リスト 5 では引数 a を for 文を用いて b に代入する処理を行っている.

リスト 5: setVector 関数のコード

```
1 void setVector(double a[DIM][1], double b[DIM][1]){
2     int i;
3     for(i=0; i<DIM; i++){
4         b[i][0] = a[i][0];
5     }
6 }
```

3.2.7 main 関数

main 関数は Euler 法を用いて連立微分方程式を計算する. 表 7 に main 関数の機能, 引数, 戻り値の 3 つを示す. main 関数は

表 7: main 関数の機能, 引数, 戻り値

機能	連立方程式の計算を行う
引数	なし (void)
戻り値	(int)0

リスト 6 に main 関数のソースコードを示す.

リスト 6: main 関数のコード

```
1 int main(void){
2     double h = 0.1;
3     double lim=1.01;
4     double a=0.7;
5     double b=0.4;
6     double c=0.09;
7     double d=0.06;
8     double y10=10;
9     double y20=10;
10    double step;
11    int i;
12
13    // 初期値を宣言するベクトルの配列
14    double initVector[DIM][1] = {{y10}, {y20}, {y10*y20}};
15    // ベクトル変換用配列
16    double transVector[DIM+1][1];
17    // 連立微分方程式のパラメータの配列
18    double weightMatrix[DIM][DIM+1] = {{0, a, 0, -c},
```

```

19         {0,0,-b,d},
20         {0,0,0,0}
21     };
22
23     // 計算用配列
24     double yiVector[DIM][1];
25     double tmpVector[DIM][1];
26     //結果表示用配列
27     double resultVector[DIM][1];
28
29     // 初期値表示
30     #ifdef STDOUT
31     printf("step = %.2lf\n",0.1);
32     #else
33     printf("%.1f,%.1f,%.1f",0.0,initVector[0][0],initVector[1][0]);
34     #endif
35
36     for(i=0;i<DIM;i++){
37         #ifdef STDOUT
38         printf("y%d = %.1f\n",i,initVector[i][0]);
39         #endif
40     }
41     printf("\n");
42
43     // 初期値を計算用配列にセット
44     setVector(initVector,yiVector);
45
46     for(step=h;step<=lim;step+=h){
47         //ベクトル変換
48         transformVector(yiVector,transVector);
49         // 微分係数を計算
50         multipleMatrix(weightMatrix,transVector,tmpVector);
51         scalerVector(tmpVector,h);
52         // 1次ラグとの和を計算
53         addVector(yiVector,tmpVector,resultVector);
54
55         //結果を表示
56         #ifdef STROUT
57         printf("step = %.2lf\n",step);
58         for(i=0;i<DIM;i++){
59             printf("y%d = %.1f\n",i,resultVector[i][0]);
60         }
61         printf("\n");
62         #else
63         printf("%.1f,%.1f,%.1f\n",step,resultVector[0][0],resultVector[1][0]);
64         #endif
65
66         // y1*y2調整処理
67         resultVector[2][0] = resultVector[0][0]*resultVector[1][0];
68         // 結果を計算用配列にセット
69         setVector(resultVector,yiVector);
70     }
71     return 0;
72 }

```

リスト 6 では次に示す手順で処理を行っている。

1. ステップ数, 連立方程式のパラメータの値, 初期条件の値, ループ用変数の 4 つを宣言する (リスト 6 の 2 行目から 11 行目).
2. 初期条件を `initVector` という変数で宣言する (リスト 6 の 14 行目).
3. ベクトル変換用の配列を `transVector` という変数で宣言する (リスト 6 の 16 行目).
4. 連立方程式のパラメータを `weightMatrix` という変数で宣言する (リスト 6 の 18 行目).
5. 計算および結果表示用の配列を宣言する (リスト 6 の 23 行目から 26 行目).
6. 初期値を表示する. 本プログラムでは条件付きコンパイルを用いる. オブジェクト形式マクロで `STDOUT` が宣言されているときは標準出力に適したフォーマットで計算結果を出力する. `CSVOUT` が定義されているときは, csv 出力に適したフォーマットで出力する (リスト 6 の 29 行目から 40 行目).

7. 計算用配列に初期条件をセットする (リスト 6 の 43 行目).
8. ステップ数に応じて手順 9 から手順 14 を反復実行する.
9. ベクトルの変換を行う. ここでの変換とは n 次元ベクトル $x = (x_1, x_2, \dots, x_n)$ を $(n+1)$ 次元ベクトル $x' = (1, x_1, x_2, \dots, x_n)$ にする変換のことである (リスト 6 の 47 行目).
10. 行列の積を用いて微分係数を計算する (リスト 6 の 49 行目から 50 行目).
11. ベクトルの和を用いて Euler 法の近似計算を行う (リスト 6 の 52 行目).
12. 結果を表示する. 表示形式は手順 6 と同様に STDOUT, CSVOUT の場合分けが行われている (リスト 6 の 55 行目から 63 行目).
13. $y_1(t)y_2(t)$ の調整を行う (リスト 6 の 66 行目).
14. 次のステップの計算のために計算結果を計算用配列にセットする (リスト 6 の 68 行目).

3.3 実行結果

3.4 考察

参考文献

- [1] 国立高専機構長野高専, <http://www.nagano-nct.ac.jp/>, 閲覧日 2020 年 8 月 5 日