R 言語によるシミュレーション～ライフゲーム

参考文献　ライフゲイムの宇宙　ウィリアム･バウンドストーン、有澤誠訳、日本評論社

さて、R 言語の基本についてある程度学んできたので、そろそろ R 言語をつかったシミュレーションを行ってみよう。まずは、古くて新しい、単純で複雑怪奇な世界…ライフゲームのシミュレーションを行ってみよう。

ライフゲームとは 1970 年にケンブリッジ大学の John Hurton Conway が発明し、M. Gardner がサイエンティフィックアメリカン誌（日本では「日経サイエンス」として刊行されている）1970 年の 10 月号と 71 年の 2 月号で紹介し大ブームになったゲームである。Conway はケンブリッジ大学のコーヒールームで碁石をつかって“手で碁石を動かして”遊んでいたが、やがて碁石はコーヒールームから廊下へと発展していったそうな。当時、電子計算機の黎明期であまたのハッカーたちが当時高価だった計算機を駆使してライフゲームにいそしんでいた。それは 1974 年の「Time 誌」が「ライフゲームの大群が数百万ドルの貴重なコンピュータ時間を食ってしまっている」と苦言を呈したほどである。また、人工生命の父、C. Langton も当時、徴兵忌避の代わりに病院で働いていたが、夜中に誰もいないはずの計算機センターでライフゲームにいそしんでいたときに“何者かの気配”を感じ、その“気配”がライフゲームであることに気づき人工生命を着想するに至った…そうである。

ライフゲームの世界は深淵であり、そこに何かの法則性を見いだすことは極めて困難である。それはちょうど天体観測をするように、ライフゲームの研究はライフゲームの宇宙を観測することで研究をすすめている。そこでは、自然系と異なり「すべての相互作用は一分の隙もなくわかっている」、それなのに、そこで生じる世界は宇宙のように膨大なのである。ライフゲームの“創造主”はまぎれもなく、そのゲームをつくった人間なわけではあるが、あまりにも相互作用の組み合わせが膨大すぎで、それらを統一的に理解することができないのである。

近現代の科学は、要素に還元し要素間の相互作用を明らかにすれば、自然は理解できると“思い込んでいる”。そして、素粒子論や分子生物学が発展してきた。だが、要素に還元して相互作用を精密に明らかにすると、もしかしたら、ライフゲームの宇宙のような理解することが困難な世界へ行き着いてしまうのかもしれない。閑話休題。

ライフゲームの設定

ライフゲームな N×N のセルとして定義される。セルの状態は１（生）と０（死）の２状態をとる。そして、自分のセルの隣のセルによって生／死が決まる。自分の左右と

上下の４近傍の場合を“ノイマン近傍”、これに斜め上下を加えた８近傍の場合を“ムーア近傍”という。ライフゲームではムーア近傍（８近傍）を使う。

ルール１　自分のセルが死(0)で、近傍の８セルのうち３つが生(1)であれば、自分は生の状態に変化する。

1　1　0　　1　1　0
0　**0**　0 →　0　**1**　0
1　0　0　　1　0　0

ルール２　自分のセルが生で、周囲の８セルのうち「２つ、もしくは、３つ」が生であれば、自分は生の状態をとる。

1　0　0　　1　0　0
0　**1**　0 → 0　**1**　0
1　0　0　　1　0　0

ルール３　ルール１、ルール２のいずれにも該当しない場合には、自分の状態は死となる。

1　0　1　　1　0　1
1　**1**　1　→ 1　**0**　1
1　1　1　　1　1　1

では、このルールをプログラムしてみよう（自分でできそうな人は、是非、独力で挑戦してみてほしい。講義の進み方やプログラミングの仕方を無視してもよいので、自分でライフゲームをつくってみてください)。
ムーア近傍は３行３列の行列Ａとして、
1　1　1
1　1　1＝A
0　1　1
と表現できる。ここで”状態”とは行列Ａの真ん中つまりA[2,2]となる。そして
近傍のうち生(1)の状態がいくつあるのか？は、近傍の状態(0, 1)をすべて足し合わせれ

ばよい。行列 A の総和は
＞sum(A)
で求めることができるが、ただ sum(A)とすると自分の状態が生のときに影響してしまう。そのためにまず現在の状態、これを cur.state という名前のオブジェクトに格納しておいて、
＞cur.state <- A[2,2]
そして A[2,2]に 0 を代入して(行列への代入は“破壊的処理”なので実行すると行列 A が書き変わってしまうので、代入の前に A[2,2]をオブジェクトに代入しておく必要があるのだ)、
＞A[2,2]<- 0
この A について
＞sum(A)
とすれば、近傍の生の状態がいくつあるのかを調べることができる。

演習 以上をふまえて、ルール１からルール３をプログラミングせよ。
ルール１　自分のセルが死で、近傍のうち３つが生ならば、死→生。つまり、「自分のセルが死」は cur.state == 0 であり、かつ(&&)、「近傍のうち３つが生」　sum(A)==3 ならば「死→生」となるわけだが、そのために新しい状態を格納するオブジェクト new.state を定義しておいて new.state<-1 とすればよい。以上をまとめると、

if (cur.state == 0 && sum(A)==3)　{new.state<-1}

となるだろう。同様に考えるとルール２　「自分のセルが生」(cur.state==1)で、近傍のうち２つ、または(||)、３つが生(sum(A)==2 || sum(A)==3)ならば生(new.state<-1)となるので、これをまとめると、

if(cur.state==1 && (sum(A)==2 || sum(A)==3)) {new.state<-1}

となる。もうひとつルール３があるが、ひとまずここまでプログラミングしたところで、ルール１とルール２を組み合わせてみよう。まず、ルール１は

if (cur.state==1 && sum(A)==3)

であるが、もし現在の状態が生の場合はどうなるかをちょっと考えてみると、ルール１は適用できないが、現在の場合が生で、近傍の生の数が３ならば、どうせルール２によって状態は生→生となる。なので、ルール１で現在の状態が生か死かを考える必要がない。なので

if (sum(A)==3)

としても差し支えないだろう。

次にルール２であるが、そのための条件は“近傍の生の数が３もしくは２”である、ルール１の条件は「近傍数が３か否か」であったので、これと組み合わせてかんがえることができるだろう、つまり、

もし　近傍数が３ … ルール１
　　または（||）、近傍数が２

とすれば、ルール１とルール２の条件部を一つの if 文で記述することができる。ただし、ルール２では状態が生でなければならない。近傍数が３、または、近傍数が２で“状態が死”の場合には、状態が死の場合が条件になっているルール１では問題ないが、状態が生の場合についてのルール２の場合には、適用ができない。そこで、この条件部に加筆する、

もし　近傍数が３
　　または(||)、(近傍数が２　かつ(&&) 現在の状態が生)
ならば　{new.state<-生}

こうすれば、ルール１とルール２の条件部を１つの if 文で記述することができるだろう。そして、ルール１とルール２以外の場合、つまりルール３については、

もし　近傍数が３
　　または(||)、(近傍数が２　かつ(&&) 現在の状態が生)
　　ならば　{new.state<-生}
　　それ以外の場合(else)　…この部分が
　　{new.state<-死} …ルール３に該当

となる。少し頑張ってルール１とルール２を一つの if 文にまとめてしまったので、ルール３をこのように簡潔に表現できた。

演習 以上の説明をもとに、ライフゲームのルールをプログラミングせよ。

ヒント:

```
life.game.rule <- function(M){
A <- M
cur.state<- (現在の状態、つまり、行列 A の 2 行 2 列目の要素を代入)
A[2,2]<-0     #現在の状態を 0 にする
if ( ここに先述したルール１とルール２の条件){
new.state<-1
}
else {              #ルール 3
new.state<-0 #ルール 3
}
new.state
}
```

周期境界条件

さて、ライフゲームのルールについてはプログラミングできたが、こんどはセルについて考えなければならない。本資料ではライフゲームのセルは２次元としている。そして、ルールを適用する面積は、たてーよこ３セル、ずつである。なので、２次元のセルの“縁”にあるセルにルールを適用しようとすると、たとえば右下隅のセルにルールを適用しようとしても、セルの右側も下側も境界に面しているので３セルないためにルールが適用できない。ちょうど下図での 9 のセルにルールを適用しようとしても、

```
1  2  3
4  5  6  ?
7  8  9  ?
   ?  ?  ?
```

9 の右側も下側もセルがないので（“?”となっている箇所）ルールを適用できない。こ

のような場合によく用いられるのが、周期境界条件、である。これは、境界が“球や丸い棒”のようになっていて、右の境界をつきぬけると一番左に、下をつきぬけると上につながっているとする。

周期境界条件をつくるには、まず最下部の行の下に最上部の行を挿入する。そして、最上部の行の上に最下部の行を挿入する、すると、

7 8 9
1 2 3
4 5 6
7 8 9
1 2 3

のようになる。影線の部分が挿入された部分である。同様にして右の境界の右となりに最左の列を、左の境界の左となりに最右の列を挿入する、

9 7 8 9 7
3 1 2 3 1
6 4 5 6 4
9 7 8 9 7
3 1 2 3 1

こうすると、さきほど半分が“？”だった９の近傍もきちんと８近傍がとれていることがわかる。そして、影線がついていない中央部をみると、最初にあった行列となっていることがわかるであろう。

さて、この周期境界条件のためのプログラムを考えてみよう。まず行うべきは、入力された行列 A の大きさを計測することである。たての長さを格納するオブジェクトを ht, 横幅を格納するオブジェクトを wd とすると、

＞ht<-ncol(A)
＞wd<-nrow(A)

となる。

演習

① 行列 A の最上行の上に最下行、最下行の下に最上行を挿入した行列をつくり、これを cm.tmp のオブジェクトに格納せよ。
ヒント: 行ベクトルを組み合わせて行列をつくる関数は rbind であった。なので、
＞cm.tmp<- rbind(A の最下行, A, A の最上行)

② cm.tmp 行列の最左列の左に cm.tmp の最右列、最右列の右に cm.tmp の最左列を挿入した行列をつくり、これを cyclic.map というオブジェクトに格納せよ。
ヒント：列ベクトルを組み合わせて行列をつくる関数は cbind であった。なので、
＞cyclic.map<- cbind(cm.tmp の最右列, cm.tmp, cm.tmp の最左列)

こうして周期境界条件は表現することができた。ここで、作成した cyclic.map は行列 A よりもたて（上下）−よこ（左右）が１セルずつ、合計で２セルずつ大きくなっていることに留意する。

これで準備は整った。次に cyclic.map にルール１から３を適用して書き換えていくことにしよう。まず、cyclic.map を書き換えた結果を格納するためのオブジェクト new.A をつくろう。new.A は書き換える前の行列 A と同じ大きさの零行列（要素がすべて０の行列）とする。これには先述した matrix 関数がそのまま使えるので、
＞new.A <- matrix(0, ht, wd) となる（matrix 関数（11 ページ）を参照)。次にいよいよ A を書き換えていくことになるのだが、ルールを適用するためには 3×3 の行列を cyclic.map に適用していくことになる。A[1,1]=1 の場合、その近傍は９、７、８と３、２、そして６、４、５である。これは、cyclic.map だと、横方向が１番目（９）から３番目（８)、縦方向が１番目から３番目（６）までとなる。これはつまり、

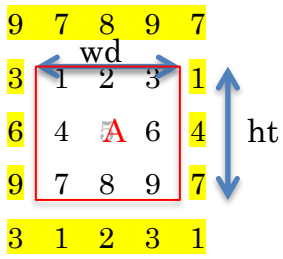
横方向：i 番目から i+2 番目まで
縦方向：j 番目から j+2 番目まで

となる。ここでたとえば、i を１から wd まで動かすとすると、A が３×３の大きさの行列の場合（下図)、i=wd 番目に相当するのは cyclic.map の３列目、j=ht 番目に相当するのは cyclic.map の３行目となる。cyclic.map は行列 A よりも左右、上下それぞれ１セルずつの２セルずつ大きくなっているので、横は wd+2 まで、たては ht+2 までとなっており、３×３の大きさの行列 A の cyclic.map の大きさは５×５になっている。よって、cyclic.map で wd 番目から wd+2 番目まで、かつ、ht から ht+2 番目までの行

列（横方向が３列目から５列目、たて方向が３行目から５行目）とは、

5　6　4
8　9　7
2　3　1

であり、これは cyclic.map の右下隅と一致していることがわかる。



なので、i については１から wd まで、j については１から ht まで動かせば、cyclic.map にまんべんなくルール１から３を適用できることになる。

　たとえば、A[1,1]=1 を書き換える場合、A[1,1]の近傍は cyclic.map でいえば、cyclic.map[1:1+2, 1:1+2]（１列目から１＋２列目、１行目から１＋２行目）、つまり、

9　7　8
3　**1**　2
6　4　5

となる。また、A[1, 2]=2 であれば、cyclic.map[1:1+2, 2:2+2]、つまり、

7　8　9
1　**2**　3
4　5　6

となる。つまり、A[i, j]の要素を書き換えるには、cyclic.map[i:i+2, j:j+2]を用いることになる。

　A[i, j]の要素の書き換えは、30 ページの演習問題でつくった life.game.rule 関数を用いればよいので、この関数に cyclic.map[i:i+2, j:j+2]をわたせば、A[i, j]の新しい要素を得ることができる。なので、この新しい要素を new.A[i, j]の新しい要素とすれば、A[i,

j]を書き換えていくことができる。

演習　行列 A の書き換えの部分をプログラミングしてみよう。以上をまとめると、以下のようになる。() 内を考えなさい。

```
life.game.1step<- function(A){
 ht<-ncol(A)
 wd<-rcol(A)
 cm.tmp<- rbind(この部分を考えなさい) #行列 A の上下に最下行と最上行を重ねる
  cyclic.map<-cbind(この部分を考えなさい)#行列 cm.tmp の左右に最右列と最左列
                                        を加える
 new.A <- matrix(0, nrow=ht, ncol=wd)
 for (i in 1:wd)
    for(j in 1:ht)
       new.A[i, j] <- life.game.rule(cyclic.map[(この部分を考えなさい)])
new.A
}
```

以上で、ライフゲームの本体部分は完成した。実行のためには、以下の関数をもちいる。

```
play.life.game <- function(A, step=20, pause=TRUE){
 for (i in 1:step) {
 image(A, axes=FALSE)
 if(pause) readline (prompt="Press <Return> to continue.")
 next.A<- life.game.1step(A)
 title(main=paste("Step:", i))
 A <- next.A
 }
}
```

演習　これまでつくってきた、３つの関数をまとめて１つのファイルとして、lifegame.R のファイル名で保存しなさい。(拡張子が.R になっていることに注意、テキストエディタだと lifegame.R.txt となっている場合があるので、その場合には.txt の部分を削除すること)

以上ができたら、初期状態をつくろう。

演習　以下に説明する方法により、glyder.txt ファイルをつくれ。

まず、data.list 関数を定義する。

```
data.list <- function(d)
{
cat(paste(t(d), c(rep("¥t", ncol(d)-1), "¥n")), sep="")
}
#このプログラムは http://aoki2.si.gunma-u.ac.jp/R/data-list.html を使いました。
```

次に、

```
＞tmp<-matrix(0, nrow=10, ncol=10)
＞t_ls<-c(2,3, 3,2, 4,2, 4,3, 4,4)
＞th<-c(1,3,5,7,9)
＞for (i in th){
  tmp[t_ls[i], t_ls[i+1]]<-1
  i<-i+2}
＞data.list(tmp)
```

より以下と同じ行列をつくる；

```
0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 1 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

この行列をメモ帳のようなテキストファイルにペーストして glider.txt として保存する。

そして、
＞source(“ディレクトリ名/lifegame.R”)
ファイル→ソースコードを読み込み→lifegame.R を選択する（このときに、lifegame.R のディレクトリ名を確認）
＞map<-as.matrix(read.table(“/(ここに確認したディレクトリ名)/glider.txt”))
＞play.life.game(map)
とすると、ライフゲームが実行される。
#実行されない場合は glider.txt のディレクトリ名をすべて記述する。

実行結果は別のウィンドウにグラフが表示され、そして、コマンドラインのウィンドウには、下記のように表示される。

Press <Return> to continue.

リターンキーを押すことにより 1 ステップごとに実行することができるが、面倒な場合には play.life.game()を実行するときに、pause=FALSE として、

＞play.lfe.game(map, pause=FALSE)

とすればよい。現在の設定だと 20 ステップまでしか計算しないので、play.life.game<-function(A, step=… のところを 100 や 300 などにするとより長いステップ数の計算を行うことができる。

```
play.life.game <-function(A, step=ここにステップ数, pause=TRUE){
        for (i in 1:step){
          image(A, axes=FALSE)
          if (pause) readline(prompt="Press <Return> to continue.")
          next.A <- life.game.1step(A)
          title(main=paste ("Step:", i))
          A<- next.A
          }
          }
```

ライフゲームの宇宙

この課題のプログラムとシミュレーションはちょっと小規模すぎて、ライフゲームの宇宙のような世界を楽しむには少しものたりない。なので、もう少し大きな初期状態をつくってみよう。R 言語に少し慣れてきたひとは、自分なりにランダムな初期状態をつくるプログラムをつくってみるとよい。たとえば、

```
rand.field<-function(size, rand){
tmp_tmp<-NULL
for (i in 1:(size * size)){
 ran<-sample(100, 1)
 if (ran < rand)
            {tmp<-1}
 else
            {tmp<-0}
            tmp_tmp <-c(tmp_tmp, tmp)}
             tmp_tmp <- matrix(tmp_tmp, nrow=size, ncol=size, byrow=TRUE)
tmp_tmp
}
```

とすると size×size の大きさの初期状態をつくることができる。また rand は全体の rand パーセントが生の状態であることを示している。たとえば、

```
＞play.life.game(rand.field(40, 10), 1000)
```

とすれば、40×40 の大きさの、全体の 10 パーセントが生の状態の初期状態をランダムに生成して、1000 ステップまでライフゲームを行うことができる。

Life Game のサンプルコード

```
rand.field<-function(size, rand){
tmp_tmp<-NULL
for (i in 1:(size * size)){
 ran<-sample(100, 1)
 if (ran < rand)
{tmp<-1}
 else
{tmp<-0}
 tmp_tmp <-       c(tmp_tmp, tmp)}
  tmp_tmp <- matrix(tmp_tmp, nrow=size, ncol=size, byrow=TRUE)
tmp_tmp
}

data.list <- function(d)
{
 cat(paste(t(d), c(rep("¥t", ncol(d)-1), "¥n")), sep="")
}

life.game.rule<-function(M){
N<-M
cur.state<-N[2,2]
N[2,2]<-0
if ((sum(N) == 3)
    ||(sum(N) == 2 && cur.state == 1))
    {
       new.state<-1
    }
 else
 {
  new.state<-0
 }
```

```
 new.state
 }

life.game.1step <- function(A){
 ht <- nrow(A)
 wd <- ncol(A)
 cm.tmp<-rbind(A[ht,], A, A[1,])
 cyclic.map <- cbind(cm.tmp[,wd],cm.tmp,cm.tmp[,1])
 new.A<-matrix(0, nrow=ht, ncol=wd)
 for (i in 1:ht)
       for (j in 1:wd)
       new.A[i,j]<- life.game.rule(cyclic.map[i:(i+2), j:(j+2)])
  new.A
  }

  play.life.game <-function(A, step=500, pause=TRUE){
       for (i in 1:step){
        image(A, axes=FALSE)
        if (pause) readline(prompt="Press <Return> to continue.")
        next.A <- life.game.1step(A)
        title(main=paste ("Step:", i))
        A<- next.A
        }
             }
```

発展課題

ライフゲームのルールを変えると、異なった時間発展をみせるようなモデルとなる（ルールの定義の部分を変えればすぐにできるので、やってみると面白い）。また、ライフゲームでは“生／死”(0 / 1)の 2 状態であったが状態数を増やすこともできる。

参考：“グライダー銃”をキーワードにしてサーチしてみるとよい。

セルラーオートマトンを用いたモデルの構築

ライフゲームのようにとなり近所との“相互作用”により、状態が変わっていく系をセルラーオートマトンとよぶ。ライフゲームは碁盤や将棋盤のようなマス目の各々に状態があり、それらが変化していくことになるが、この“マス目”のことを「セル」よぶ。そして、そしてルールにより状態が変化していくモデルは「オートマトン」とよばれる。セルラーオートマトンとは“マス目”がルールにそって変化していく系のことである。

こうした「となり近所の相互作用による状態の変化」をもちいてモデル化を行ってみよう。たとえば「草食動物と肉食動物からなる生態系」を考えてみよう。

モデル化を行うために必要なこと

何らかのモデルをつくってモノをかんがえること…こうした“技術”は研究者の特殊技能のように思うかもしれないが、私たちは日常生活でよくおこなっていることである。たとえば、友人や知人を私たちは「モデル化」している。そしてそのモデルに基づき「こんなことを云ったら気分を害するかな？」、とか、なにか困ったことがあったときに「あの人ならば相談にのってもらえるかな」と「シミュレーション」をして、その結果に基づいて行動する。そしてシミュレーションの結果と、実際に行った行動の結果がズレている場合には「あんな人だと思わなかった！」と思ったり、「あの人ってこんな一面があるんだ…」と発見したりして“検証”を重ねて自分のモデルを修正していく。

友人や知人を「モデル化」するときに私たちはどうしているだろう？相手のすべてを知ることなどできないし、一挙手一投足のすべてを観察しているわけでもない(だろう)。たいてい、ざっくりと「いいひと」とか「気難しいひと」のように、大雑把かつ感覚的に性質をとらえてモデル化を行う。これは数理モデルを構築する場合でも同様で、まず重要なことは「モデル化を行う対象を感覚的にとらえる」ことである。そのためには実験をしたり、実験家から話を聞いたり、直接に対象を見にいったり，etc… そして対象が「ざっくりと云ってどんなモノか」を“掴むこと”がなによりも重要である。

ここで“やってはいけないこと”は対象を直感的にとらえることをせずに“ただ写生すること”である。例えば、渋滞をモデル化する場合に、渋滞は“車”によるものであるからといって、車、道路、運転者、天候、などなどといくらでも詳細にモデル化する、まさに写し取るようにモデル化はできるだろうが、それでは縮尺が一対一の地図をつくる努力をするようなもので、詳細になればなるほど、対象の複雑さや理解しにくさを忠実に写し取ることになってしまい、どんどん“わけのわからない”モデルになってしまい、そのモデルを理解するためのモデルが必要となってしまうであろう。

さて、では草食動物と肉食動物による生態系について考えて（妄想して？）みよう。そのため肉食動物を“１”、草食動物を“０”とする。肉食動物はいつもおなかが減っているわけでもないだろうから、肉食動物に出会ったらすぐに食べられてしまうわけではない（であろう）…

演習

①肉食動物の「おなかの空きぐあい」をモデル化するためにはどうしたらよいか考えてみよ。

ヒント：どのようにルールをデザインしたらよいだろうか？

②　①で考えたルールをプログラミングしてシミュレーションを実行させてみよ。

```
lv.rule<-function(M){
 N<-M
 cur.state<-N[2,2]
 if (cur.state == 0 && sum(N) >= 8)
 {
        new.state<-1
 }
 else
 {
        new.state<-0
 }
 if (cur.state == 1 && sum(N) <= 8)
 {
        new.state<-0
 }
 else
 {
        new.state<-1
 }
 new.state
```

```
}
```

R グラフィクス

この生態系のシミュレーションでは、草食動物と肉食動物の個体数の変化が気になる。そのため、ここで少し R 言語によるグラフィクスについて触れておくことにしよう。

R 言語では X11 というグラフィクス画面を利用して描画を行う。まず plot 関数を使用して、点をプロットしてみよう。

```
＞x<- c(1,2,3,4,5)
＞y<-c(2,3,4,5,9)
＞plot(x,y)
```

plot はさまざまな引数をとる。たとえば、type 引数の場合は、

type=”p” … 点を用いてプロット
type=”l” … 線でつなげる場合　（1 ではなくて L の小文字 l）
type=”b” … 点と線を用いてプロット
type=”n” … 枠のみでデータをプロットしない

演習

① ＞plot(x,y,type=”ここに引数をいれる)により描画をおこなってみよ。

② これでセルラーオートマトンをもちいて生態系のモデル化を行うことができた。このモデルを用いてシミュレーションを行いなさい。このモデルでは“ルール”の定義によりさまざまな状況をも考えることができる（例えば、肉食動物は周囲（近傍に草食動物が 0 の場合に死亡し、そのかわり草食動物が増加する、など)。

さまざまな状況においてのシミュレーションを行い、個体群の動態がどのように変化するかを調査し、その結果をまとめなさい。

ヒント：草食動物と肉食動物の生態系のプログラムは、例えば、以下のようになる。

```
lv.rule<-function(M){
 N<-M
```

```
 cur.state<-N[2,2]
 if (cur.state == 0 && sum(N) >= 8)
 {
        new.state<-1
 }
 else
 {
        new.state<-0
 }
 if (cur.state == 1 && sum(N) <= 8)
 {
        new.state<-0
 }
 else
 {
        new.state<-1
 }
 new.state
}

lv.1step <- function(A){
  ht <- nrow(A)
  wd <- ncol(A)
  cm.tmp<-rbind(A[ht,], A, A[1,])
  cyclic.map <- cbind(cm.tmp[,wd],cm.tmp,cm.tmp[,1])
  new.A<-matrix(0, nrow=ht, ncol=wd)
  for (i in 1:ht)
        for (j in 1:wd)
    new.A[i,j]<- lv.rule(cyclic.map[i:(i+2), j:(j+2)])
    new.A
    }

play.lv <-function(A, step=200, pause=TRUE){
```

```
Y<-0
X<-0
msize<- prod(dim(A))
      for (i in 1:step){
 image(A, axes=FALSE)
       if (pause) readline(prompt="Press <Return> to continue.")
       next.A <- lv.1step(A)
       Y <- c(Y, sum(A))
 tmp <- msize - sum(A)
 X <- c(X, tmp)
 plot(Y, type="b")
 par(new=T)
 plot(X, type="b")
       title(main=paste ("Step:", i))
       A<- next.A
       }
            }
```