



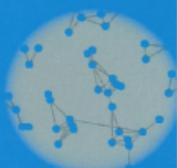
人工社会の可能性

01

人工社会構築指南

artisocによるマルチエージェント・シミュレーション入門

山影進 著



人工社会の可能性

01

人工社会構築指南

artisocによる
マルチエージェント・シミュレーション入門

目次

まえがき

注意事項

第0部 人工社会をもっと身近に

第0章 人工社会を作ろう

0.0 身近になった人工社会

0.1 人工社会のバリア・フリー化をめざして

0.2 働って慣れよう

0.3 この本の構成と内容

0.4 この本の使い方

0.5 いよいよ始めるときです

第1章 なぜ人工社会を作るのか

1.0 人工社会はおもしろい

1.1 人工社会の登場

1.2 部分に注目しながら全体を捉える

1.3 基礎にあるマルチエージェント・シミュレーション

1.4 人工生命も作れます(早く人工社会を作りたい人はとばしてかまいません。)

1.5 ネットワークも作れます(早く人工社会を作りたい人はとばしてかまいません。)

第2章 マルチエージェント・シミュレータartisocの登場

2.0 artisocに初めて触れる

2.1 人間に優しいシミュレータの誕生

2.2 シミュレータやサンプル・モデルなどの取り込み

2.3 artisocのモデルを動かす

2.4 シミュレーションの条件を操作する

2.5 中を覗いてみる(早く人工社会を作りたい人はとばしてかまいません。)

2.6 artisocを終了する

2.7 自自分で作りませんか

Column 出力一覧

第1部 モデル作りの基本を身につける

第3章 シミュレーションの準備をする

3.0 準備をすると後が楽です

3.1 シミュレーションの3要素

3.2 artisocモデルの基本

3.3 モデルの枠組みを作る

3.4 シミュレーションの過程を見るための準備をする

3.5 作業が正しかったか確認する

第4章 エージェントを動かす

4.0 ルールが肝腎です

4.1 エージェントの自律性とはどういうものか?

4.2 エージェントのルールエディタを開く

4.3 「動かす」ルールを書き込む

4.4 いよいよ実行

4.5 動かすルールや設定変更に慣れる

第5章 エージェントに判断させる

5.0 行動の選択が自律性の根本です

5.1 本格的な「自律的な行動」とは何か

5.2 エージェントに何をさせたいのかを図ではっきりと表す

5.3 「場合分け」を図示して、正しく理解する

5.4 「場合分け」ルールでエージェントに判断させる

5.5 「場合分け」のいろいろな技法

Column 大小関係を表す演算子、論理関係を表す演算子

第6章 エージェントに周囲の環境を調べさせる

6.0 観察が肝腎です

6.1 周囲の状況に応じた「自律的行動」

6.2 自分の周りのエージェントたち

6.3 新しい変数をエージェントに追加する

6.4 周囲にいる人と立ち話をさせる

6.5 パラメータを変えてみる

6.6 モデルを複雑にしよう

6.7 1種類のエージェントだけに注目する

6.8 「もっと現実的に」はどれだけ本質的か

閉話休題 酔っぱらいは自律的な主体なのか

第7章 モデルの設定値をモデルの外部から操作する

7.0 「神様」になりました

7.1 モデルの条件を簡単に設定し直したい

7.2 エージェント数を自由に変える

7.3 モデルの中でエージェントを生まれさせる

7.4 モデルの中のパラメータを簡単に変える

7.5 エージェントを生まれさせて、初期配置する

Column ツリーの中の変数とルールの中の変数

第8章 シミュレーションの過程をいろいろ出力したり、管理したりする

8.0 見ているだけが能じゃない

8.1 シミュレーションの経過をもっと知るには

8.2 モデルの中で集計させる

8.3 時系列グラフの賢い利用法

8.4 自動的にシミュレーションを終了させる

8.5 終了時にartisocに一仕事させる

Column Universeのルールエディタが複雑なのはなぜ?

第9章 格子型空間の構造を活用する

9.0 空間に「格子」の存在を想定しましょう

9.1 ほんとうは「格子」になっていない

9.2 空間は実数的だった

9.3 「格子」があるかのようにエージェントを動かす

9.4 格子を利用するモデルの外枠を作る

9.5 格子を意識したルール表現を使う

閑話休題 見方を変えると周りも変わる

第10章 第1部の修了を記念して：シェリングの「分居モデル」を作る

10.0 artisocの有り難さをじっくりと味わいましょう

10.1 分居モデルとは

10.2 コンピュータのなかの分居モデル

10.3 若干の補足

10.4 シェリングの分析を再現する

Column 変数の型

第2部 人工社会の発想と技法に慣れる

第11章 状況に応じた行動の選択肢を増やす

11.0 イフ文の使い方をマスターしよう

11.1 複雑な場合分け

11.2 ターミナル駅の通勤客の流れ

11.3 モデルの枠組みを作る

11.4 ルールを書き込む(その1)

11.5 ルールを書き込む(その2)

Column ルールを見やすくミスを少なくするためにも

第12章 エージェントの属性を豊富にする

12.0 「量より質」の豊かさを求めて

12.1 属性の「一目瞭然」化

12.2 人工国家の人口と経済

12.3 経済水準を色で表す

12.4 もっと微妙な色で属性を表す

12.5 エージェント種が異なるのか、属性が異なるのか

Column 予約語とその勝手な使用の禁止

第13章 周囲のエージェントから影響を受ける

13.0 周囲の影響から逃れられません

13.1 周囲のエージェントを詳しく調べる

13.2 病気の流行をモデル化する

13.3 「フォ・イーチ文」の登場

13.4 病気の流行を把握する

13.5 現実の流行現象に近づける

第14章 特定のエージェントから影響を受ける

14.0 誰でもよいわけではありません

14.1 エージェントをピックアップする

14.2 結婚相手をさがすモデルを作る

14.3 ルールの大枠

14.4 メスのルール

14.5 オスのルール

第15章 他のエージェントに働きかける

15.0 他者の状態を変える

15.1 積極的な働きかけ

15.2 牧羊犬の仕事をモデル化する

15.3 羊たちの行動

15.4 牧羊犬の行動

15.5 作業を完了する時間を調べる

15.6 モデル作りを工夫する(とばしてもかまいません。)

閑話休題 働きかけは「禁じ手」か「一の手」か

第16章 エージェントへの究極の働きかけ

16.0 エージェントそのものに作用できます

16.1 エージェント自体への働きかけが可能です

16.2 「殺す」は「消す」

16.3 「産む」は「創る」

16.4 個体数変動を調べる

16.5 過ぎたるは及ばざるがごとし

16.6 自然の生態系に近いモデルをめざして

第17章 非対称的な相互作用を複雑化する

17.0 そう簡単には従わない

17.1 一方的関係から非対称的かつ双方的な関係へ

17.2 幼稚園の先生の苦労をモデル化する

17.3 状況の把握を容易にする

17.4 暴走族の取り締まりをモデル化する

17.5 取り締まりの効果を調べる(とばしてもかまいません。)

17.6 命令よりは同調が重要?

第18章 空間を「場」として利用する

18.0 空間には意味があります

18.1 「場」や「勾配」は難しくない

18.2 空間変数を利用する

18.3 深海魚の泳ぐ海

18.4 深海魚の行動

18.5 空気感染をモデル化する

第19章 同期問題に注意する

19.0 シミュレーションの中の時刻と時間を意識しましょう

19.1 シミュレーションの中の時刻と時間

19.2 なぜ「同期」が重要なのか

19.3 artisocは何をやっているのか

19.4 最後にまとめて状態を変える

19.5 過去の状態を参照する

第20章 第2部の修了を記念して: コンウェイの「ライフゲーム」を作る

20.0 artisocのパワーを実感できます

20.1 ライフゲームとは

20.2 ライフゲームのモデル化

20.3 セルを貼り付ける

20.4 ライフゲームのロジック

20.5 自動的に終了させる

第3部 本格的な人工社会をめざす

第21章 エージェントを空間上で複雑に動かす

21.0 動かし方にはいろいろあります

21.1 動かし方のレパートリー

21.2 モデルの大枠

21.3 速さや方向を変えながら動く

21.4 ループなしの空間で移動する

21.5 目的地をめざす

21.6 格子空間をランダムに移動する

21.7 追いかける

21.8 逃げる

21.9 追いかけ続ける

21.10 座標系に関連づけて移動する(とばしてもかまいません。)

Column エージェント集合の中のエージェントの並び方

第22章 属性を文字列で表す

22.0 質的な属性を質的なままで操作できます

22.1 文字列で表現するとは

22.2 文字列型の変数に慣れる

22.3 文字列型変数も演算(操作)の対象になります

22.4 複雑な属性を異なった色で表す

第23章 属性を複雑に操作する: アクセルロッドの文化変容モデル(簡略版)を作る

23.0 周囲の状況に応じて、文化が複雑に変化します

23.1 文字列の書き換えを文化変容に対応させる

23.2 エージェントの初期配置

23.3 周囲のエージェントの属性に応じて自分の属性を変える

23.4 オリジナルとの違い

23.5 属性を直接マップに示す

23.6 全体の様子を把握する

Column 色の秘密

第24章 エージェントを複雑な条件で選別する

24.0 まとめた上で考えよう

24.1 エージェント集合の演算に慣れよう

24.2 友達の輪

24.3 友人を作ろう

24.4 友達になれば不即不離

24.5 友人を線で結んで明示的に示す

第25章 エージェントを識別して複雑な関係にする

25.0 相手が違えば、関係も変わります

25.1 顔見知り社会の複雑な関係に注目する

25.2 個体識別はIDで

25.3 多数の変数を配列にまとめる

25.4 同級生の関係をモデル化する

25.5 複数の関係を視覚化する

25.6 配列の扱いに慣れる

Column フォ・イーチ文とRemoveAgt()の使用上の注意

第26章 繰り返しの多いルールをすっきりと: レイノルズのボイド・モデル(2次元版)を作る

26.0 自分仕様のartisocにしよう

26.1 ルールをまとめるメリット

26.2 ポイド・モデルは簡単に作れる

26.3 ユーザ定義関数を作って、障害物をばらまく

26.4 ユーザ定義関数を作って、前方を調べさせる

26.5 障害物を避けるトリのルール

26.6 ユーザ定義関数を使い回す

第27章 さまざまな技法を組み合わせる：ランダム・ネットワークを生成する

27.0 量から質へ転化します

27.1 ランダム・ネットワークの基礎

27.2 ランダムにリンクをはっていく

27.3 ランダム・ネットワークのクラスターに注目

27.4 クラスターを区別して表す

27.5 全体像の特徴を把握する

第28章 多様な空間構造を利用する

28.0 格子や2次元にこだわることはありません

28.1 空間構造の多様性

28.2 六角モデルとは

28.3 帝国モデルの概要

28.4 帝国モデルのルール記述

28.5 レイヤとは

28.6 人事モデルの概要

28.7 人事モデルのルール記述

Column エージェントを消す三つの関数

閑話休題 現代人(?)はゼロから数える

第29章 過去を参照する：アクセルロッドのゲーム戦略選手権モデル(簡略版)を作る

29.0 過去の履歴を現在の選択に結びつけます

29.1 過去の状況を人工社会に組み込む

29.2 「囚人のジレンマ」ゲームと戦略の優劣

29.3 繰り返し「囚人のジレンマ」戦略コンテストの概要

29.4 ルールの大枠

29.5 6つの戦略をルール化する

29.6 利得を集計する

29.7 戦略のパフォーマンスを比較する

Column 繰り返し文いろいろ

第30章 エージェントに学習・適応させる：ゲーム戦略選手権モデルを発展させる

30.0 エージェントを賢くする第一歩です

30.1 学習・適応エージェントへ「はじめの一歩」

30.2 繰り返し「囚人のジレンマ」モデルの時間の重層化

30.3 「模倣による学習」のルール化

30.4 戦略の優劣を視覚化する

30.5 「適者生存」のルール化

Column 割り算こわい

第4部 研究・実務のツールにする

第31章 実行順序を制御する

31.0 時の流れに流されないように

31.1 実行順序は意外な盲点です

31.2 ステップ内の実行順序

31.3 初めと終わりの特別なステップ

31.4 ステップをまたいだ実行順序制御のテクニック

第32章 モデルのミスをチェックする

32.0 モデルが何かおかしいときは

32.1 ミスのいろいろなタイプ

32.2 モデルが動かない

32.3 思ったとおりに動かない

32.4 変数の値をもっと詳しくチェックする

32.5 より効率的に変数の値をチェックする

32.6 思ったとおりに動いている?

Column 実数と擬似実数

第33章 画像をマップに表示する

33.0 見た目も大切です

33.1 見栄えも理解を助けます

33.2 背景を表示する

33.3 エージェントを画像で表す

33.4 状況に応じて画像を切り替える

33.5 初期値を読み込む

第34章 空間変数を活用する

34.0 空間を利用するとモデルに深みが出来ます

34.1 空間変数をもっと活用しましょう

34.2 一面に雪を降らせる

34.3 航跡を描く

34.4 DLA(拡散律速凝集)をつくる

34.5 地形図のデータをマップに反映させる

34.6 地形図の「勾配」をエージェントの行動に反映させる

第35章 実験のための道具を活用する

35.0 作ったあとも重要です

35.1 作ったモデルを使いこなす

35.2 シミュレーション実行中に手を加える

35.3 亂数シード値を使って偶然の要素を排除する

35.4 いろいろな乱数

35.5 連続実行

- (1) 終了条件の設定
- (2) 出力のための準備
- (3) 連続実行の条件指定

Column 英語圏で使用する際の注意

第36章 ログ機能とファイル入出力関数を活用する

- 36.0 備えあれば臺いなし
- 36.1 ログ機能を使いましょう
- 36.2 ログを記録する
- 36.3 ログを再生する
- 36.4 ログをとることのメリット
- 36.5 ファイル入出力関数もを使いましょう
- 36.6 入力したいファイルを作成しておく
- 36.7 外部ファイルをモデルに読み込ませる
- 36.8 実行結果を出力する

第37章 シミュレーションを疑ってみる

- 37.0 謙虚さが飛躍に繋がります
- 37.1 疑う余裕を常に持とう
- 37.2 ミスを疑う
- 37.3 モデルの構造を疑う
- 37.4 モデルの挙動を疑う
- 37.5 「よい」モデルかどうか疑う
- 37.6 現実との対応を疑う
- 37.7 応用は慎重に
- 37.8 健全な懷疑に支えられて人工社会は発展します

閑話休題 KISSしたら「ヤッコー」と蹴りを入れられた?

第38章 人工社会の対象と方法を俯瞰する

- 38.0 人工社会には大きな可能性が広がっています

38.1 対象と方法

38.2 ムラ・クニ・地球、そしてその間隙

38.3 システム・モデル・シミュレーション

38.4 人工社会の得意技

38.5 未開拓の沃野へ

読書案内——高みをめざして

◎高みをめざす前に、裾野を広くしておこう(あるいは単に気分転換に)

◎人工社会についてもっと知ろう

◎複雑な人工社会の構築に向けて

◎さまざま分野でartisocを使ってみよう

◎土台をしっかりと

◎汎用マルチエージェント・シミュレータの例

◎蛇足

作成モデルの概要

著者・執筆協力者

まえがき

本書の最も重要な目的は、人工社会という新しい社会分析の発想と方法を、今までシミュレーションにもコンピュータ・プログラミングにも縁がなかった人々にも触れてもらい、なじんでもらうことです。具体的には、2006年春にリリースされたマルチエージェント・シミュレータartisocの基本技法を解説しながら、人工社会を構築する方法を解説することが中心となっています。

人工社会構築をめざして開発されたartisocですが、汎用のマルチエージェント・シミュレータという性格上、セルオートマトン、その応用としての人工生命、あるいは現在急速に一般の注目を浴びるようになったネットワークなどのモデルもartisocを用いて作ることができます。その意味では、さまざまな分野で、マルチエージェント・シミュレーションに関心があるものの、プログラミングの敷居が高すぎて手をこまねいていた人々にとっても、本書はモデル作りの解説書になっているのではないでしょうか。

本書は、科学研究費補助金学術創成研究費による「マルチエージェント・シミュレータによる社会秩序変動の研究」(平成15～19年度)の成果の一部です。この研究費により、KK-MASをプロトタイプにしたartisocの開発が可能になりました。本書は研究の成果ではありませんが、研究ツールとして開発されたartisocを紹介するという意味で、成果と言っても許してもらえるでしょう。

artisocの開発主体である株式会社構造計画研究所の関係者、特に服部正太、木村香代子、玉田正樹の3氏と、ユーザ側から開発に関わってきた関係者、特に田中明彦、原田至郎、田村誠の3氏に深謝します。本来ならば、阪本拓人、鈴木一敏、保城広至、光辻克馬、山本和也の皆さんにはいくら感謝しても感謝し足りないのですが、本書の事実上の共同執筆者として私と連帯責任を負ってもらうことにしました。実際、本書のコラム全てと第4部のほとんどの章は、彼らの手による文章です。他の部分は私の文章ですが、彼らのメモや原稿を参考にしたり、彼らの容赦ないコメントにしたがって幾度となく書き直したりしたものです。もし本書がartisocに初めて接した読者に分かりやすく書かれているとしたら、それは初心者の私がartisocを学習した過程が本書の構成や文章に反映しているからに違いありません。

本書がシリーズ「人工社会の可能性」の第1巻として日の目を見るに至ったのには、早山隆邦氏のご尽力が欠かせませんでした。長年編集者として、多くの研究者に「社会科学の冒険」をさせてきた氏のことですから、このくらいの冒険は平気なのかも知れませんが、いつもながらのご厚意に感謝いたします。

2006年11月 山影 進

注意事項

本書で使用しているartisoc textbookは、添付CD-ROM^[1]のものと同一だが、構造計画研究所で随時アップデートしているので、異なるバージョンでは仕様が若干異なる場合がある。artisocのサポートについては、MASコミュニティ(<http://mas.lke.co.jp>)を参照すること。

本書ではartisocをJava 2 Standard Edition(J2SE)5.0の上で動かしている。Java1.4.2上で動かした場合には背景色が青になるので、artisocのウィンドウが本書で例示している図と見かけが異なる。

出力マップ上のエージェントの表示色(固定色)やマーカーなどや時系列グラフの線の太さや色はartisocが自動的に選ぶが(デフォルト設定)、適宜、好みに応じて、設定し直してかまわない。本書でも、デフォルトと異なる設定をしている箇所がある。

印刷上の技術的理由により、一部の図はartisocウィンドウの実際の色と異なっている場合がある。

誤植やartisoc textbookのバージョンアップによる記述の訂正など、本書の内容の修正・更新は、随時、山影研究室ウェブサイトの教科書ページ(<http://citrus.c.u-tokyo.ac.jp/artisoc/textbook>)に掲載する。

シミュレータおよびモデル・ファイルなどに関する注意事項(必ず読んで下さい)

本書に添付されているCD-ROMには、株式会社構造計画研究所から提供されたシミュレータartisoc textbookのバージョン1.0、数種のサンプル・モデルならびに日本語版と英語版のマニュアル、ヘルプ機能が記録されています。株式会社構造計画研究所が推奨するartisoc textbookの動作環境としては、Microsoft Windows XP/2000またはApple Mac OS XをOSとするパソコン上で、Java1.4.2以上ならびにAdobe Acrobat Reader 7.0以上がインストールされていることが必要です。なお、artisoc textbookに関するサポートは、MASコミュニティ(<http://mas.lke.co.jp>)で行なっ

ています。

artisoc textbookは、基本的にはartisoc academicのバージョン1.0と同等品であり、後者に備わっている機能を全て使用することができます。ただしartisoc textbookには、書き込めるルールの総計が200行までである、入出力できるテキストファイルのファイル名が「input.txt」と「output.txt」の2つのみであるという制限がかかっています。もちろん、この制限は、本書で学習する範囲では何ら支障ありません。

本書の各章で使用・作成するモデル・ファイル、練習問題の解答例としてのモデル・ファイルおよび第4部で使用する外部ファイルについては、添付されているCD-ROMではなく、東京大学 山影研究室のウェブサイト上にある本書用のページ(<http://citrus.c.u-tokyo.ac.jp/artisoc/textbook>)からダウンロードすることができます。

読者によるartisoc textbookのコンピュータへのインストールは、読者本人が所有または使用するコンピュータ1台のみに限らせていただきます。添付のCD-ROMの管理は、読者本人が責任をもって適切に行って下さい。また、artisoc textbookで構築したモデルに基づいて研究発表をする際には、その旨を発表資料等に明記して下さい。

なお、いかなる場合もインストールによって生じる対象コンピュータの記憶装置に記憶されたプログラムおよびデータの破損、損傷、変更、消失についての責任はいっさい負えませんのでご了承願います。その他、artisoc textbookのご使用、あるいは本書に書かれているモデルや技法のご使用によって、何らかの損失や影響が生じたとしても、本書の著者、出版者ならびに株式会社構造計画研究所はいっさいの責任を負えません。以上のことをご了承していただいた上で、artisoc textbookをご使用願います。

-
1. CD-ROMの内容を以下のURLからダウンロードすることができます：<http://mas.lke.co.jp/books/>。
書籍コード“8425”を入力し、本書に関する質問に答えてください。答えが正しければダウンロードURLが表示されます。

第0部 人工社会をもっと身近に

マルチエージェント・シミュレーションという技法に基づく人工社会は、注目されながら、あまり浸透してきませんでした。しかしこれからは、人工社会は少数の研究者の独占物ではありません。パソコン・ソフトの登場で、人工社会がとても身近になりました。誰でもすぐに体験できます。そして少しの練習時間があれば、自分でモデルを作り、シミュレーションを実行できます。この第0部では、人工社会を作ることへの関心を高めてもらい、実際に体験してもらいます。

第0章

人工社会を作ろう

0.0 身近になった人工社会

- プログラミング技法やプログラミング言語の知識は不要です
- シミュレーションがリアルタイムで見られます
- 文系の人たちに向いています
- パソコンさえあれば、1人でも学べます
- 本格的な学術研究や実務のツールにもなります

0.1 人工社会のバリア・フリー化をめざして

人工社会という社会現象の分析や理解にとってきわめて有望な方法が、欧米の社会科学の学界で注目され始めています。この方法は、マルチエージェント・シミュレーションというコンピュータ・シミュレーションの技法に基づけられています。マルチエージェント・シミュレーションについては、日本では、理工系の学問分野で急速に蓄積されつつあるものの、社会科学ではまだまだ浸透していません。

なぜでしょうか。マルチエージェント・シミュレーションが十分に理解されていない理由はおそらく、次のようにまとめることができるでしょう。

- 参考になる具体的な適用事例が乏しい
- プログラミング言語やプログラミング技法を学びたくない
- 社会科学の方法としては、シミュレーションはうさんくさい
- マルチエージェント・シミュレーションを学ぶ場がない

これらは皆もっともな理由です。そして、どれも互いに結びついています。したがって、このままで、マルチ

エージェント・シミュレーションは社会科学になかなか浸透せず、人工社会という新しい方法も普及していないかもしれません。

このような状況は、いわば「食わず嫌い」でマルチエージェント・シミュレーションが敬遠されている状況です。これを乗り越えて、マルチエージェント・シミュレーションが社会現象の分析に広範に使われるようになるように、私たちは次のことをめざしています。

- 参考になる具体的な適用事例を示す
- プログラミング言語やプログラミング技法を学ぶ必要をなくす
- 社会科学の一方法として学界に認めさせる
- マルチエージェント・シミュレーションを学べる機会を増やす

上のようなことを実現するには、何よりも、マルチエージェント・シミュレーションを簡単に実行できるソフトウェアが必要です。そこで私たちは、誰にでも抵抗なく使えるだけでなく、人工社会の本格的な研究に特に適したマルチエージェント・シミュレータの開発を進めることにしました。

0.2 做って慣れよう

そしてこの度、ようやくプログラミング言語やプログラミング技法を学ばないでも利用できる、そしてパソコンの機種にも依存しないシミュレータができました。使いやすいマルチエージェント・シミュレータをめざして数年前に開発されたKK-MASをプロトタイプにしつつ、抜本的に改良した第2世代のシミュレータです。これにより、簡単にマルチエージェント・シミュレーションを行なえる環境ができました。今までマルチエージェント・シミュレーションを敬遠してきた人たちにとっても、マルチエージェント・シミュレーションの世界を覗いてみることが容易になりました。

この新しいシミュレータは、特に、人工社会として社会現象のモデルを作りシミュレーションを実行することを念頭に置いて開発されたものです。その名も、人工社会(artificial society)を縮めたartisoc(アーティソックと発音して下さい)です。

このシミュレータartisocを使うと、マルチエージェント・シミュレーションのためのモデルが容易に作れるだけ

でなく、シミュレーションの実行も実行過程のリアルタイムでの観察も容易になりました。簡単なモデルなら数分で完成させ、ただちに実行して、見ることが可能です。大きさではありません。そして、この本の第一部で紹介しているartisocの使い方だけで、2005年ノーベル経済学賞を受賞したトマス・シェリングが考案した「分居モデル」という有名な古典的モデルを完成させることができます。それどころか、もっと洗練したモデルにすることも簡単です。

「習うより慣れよ」ということわざがあります。その趣旨を活かしながら、ここでは同じ「ならう」でも、「習う」よりは「做う」の方を重視して、「做うことで慣れよ」というふうに読み替えたいと思います。つまり、モデル作りを例題に沿って真似していくうちに、自分でモデルを作れるくらいにシミュレータに「慣れていき」、そしてマルチエージェント・シミュレーションについて「わかった」という実感を得ることをめざしています。

0.3 この本の構成と内容

この本は、artisocに準拠して人工社会の技法を学ぶことをとおして、マルチエージェント・シミュレーションの発想になじんで、自分自身の問題関心に即したモデル作りとシミュレーション実行を可能にすることをめざしています。

本体は5部構成になっています。この章を含む「第0部 人工社会をもっと身近には、人工社会とかマルチエージェント・シミュレーションとかいった言葉になじみのない人たちへの導入部です。いわば、この新しい発想に基づいた新しい技法の「お披露目」です。「第1部 モデル作りの基本を身につける」では、人工社会の(1)モデルを作り、(2)シミュレーションを実行し、(3)その過程を見る、という基本を一通りおさらいします。モデル作りでは、エージェントを相互作用させるために必要な、ごく基本的な最小限の技法を紹介します。「第2部 人工社会の発想と技法に慣れる」では、マルチエージェント・シミュレーションのさまざまな技法を紹介します。そこでは、第1部で示した人工社会の扱い方に、さまざまに肉付けていきます。「第3部 本格的な人工社会をめざす」は、artisocが備えている機能を使いこなすことによって、高度な技法を駆使して人工社会に本格的に取り組むことをめざします。第1部から第3部まで、いわば螺旋状に、そこで扱われる人工社会の技法が高度化していきます。最後の「第4部 研究・実務のツールにする」は、人工社会の技法を実務や学術研究で用いる際の要領や注意点をまとめてあります。

この本では、特に第1部から第2部まで、さまざまな技法を少しづつ積み上げていきますが、各段階での新しい技法になじんでもらうため、簡単に作れて見た目にインパクトのあるモデルを次々と作ります。その中には、マルチエージェント・シミュレーション関係(特に人工社会、人工生命、複雑系、ネットワークについて)の本には必ずと言ってよいほど紹介されているモデルがいくつも含まれています。既に触れたシェリングの分居モデルもそのひとつです。

なお、この本の全体をとおして、随所に「コラム」と題した囲み記事が置かれています。コラムのトピックは、本文中の説明としてはあちこちに出てくるがひとまとめにして説明した方がよい技法や事項を取り上げています。また、ところどころに「閑話休題」という駄文があります。休憩時間の暇つぶしといった気分で気軽に読める文体で、マルチエージェント・シミュレーションやartisocに関する話題を取り上げています。

0.4 この本の使い方

この教科書は実践を重視しています。読むだけでは決して人工社会の達人にはなれません。人工社会の技法を身につけるには、この本を自分のパソコンの横に置いて、読みながら実際にモデルを作り、シミュレーションを実行する必要があります。「畳の上の水練」ということわざがありますが、教科書を読むだけというのはまさにそれです。バイオリンの教則本をいくら読んでも、バイオリンの名手になれないと同じです。とは言っても、教則本に従って練習をしなければ名手になれません。この点でも同じです。したがって、第1部を済ませるまでは、なるべく教科書の指示どおりに、自分自身でモデルを作り、それを動かしてマルチエージェント・シミュレーションに慣れて下さい。

この本は、基本的に独習で、マルチエージェント・シミュレーションによる人工社会の構築と分析の基本をマスターできるように作られています。人工社会に関心ある学生や社会人にとって、人工社会を操る技法を教えてくれる場は現状ではまだまだ少ないと思えるからです。そのためにいくつか工夫をしてあります。

- 各章をなるべく「小さなまとまり」にして、だいたい30分以内で一通りさらうことができるよう配慮しました(したがって、分量的には同じでも、後の方になるほど作業量は増える傾向にあります)。
- 短時間に簡単に作れ、しかも動かしてみると見た目に印象深いモデルを、なるべくたくさん作ってもらうことで、飽きが来ないように努めました(そのため、人工社会という名前にはふさわしくない単純なモ

デルも作ってもらいます)。

- 各章で学んだ内容の復習を兼ねて、自分の頭でモデルの作り方を考える練習になるような課題を章末に設けました(なるべく、全部の課題をやってください)。
- 有名な古典的なモデルとなるべく早い段階で作れるように、章の構成に配慮しました(それによって、何かを成し遂げたという満足感が高まるでしょう)。
- モデル作りの理屈やシミュレーションの技法についての体系的説明はコラムにまとめることによって、この本が無味乾燥のマニュアルにならないように留意しました(したがって、個々のコラムはいろいろな章の内容にまたがっています)。

独習でこの本を使う場合は、とにかく第1部の第3章から第9章までを終えて下さい。分かりにくい箇所があつても、そこで立ち止まらず、とりあえず、この本の指示に従って、モデルを作つていって下さい。そこまでで学ぶ技法でシェリングの分居モデルを作れます。実際、第1部最後の第10章では分居モデルを作ります。ノーベル経済学賞の受賞者による研究と同じレベルの研究がこれでできます。シェリングの使つた道具はチェス盤、コインそしてサイクロド、シミュレーションは時間と手間のかかる作業でした。しかし今日は、パソコンのおかげで、簡単にシミュレーションをすることができるだけでなく、シェリングの分析よりもっと高度な分析も可能です。ここまで来たらもう安心です。再び第1部を読み返してみると、以前引つかかつたり分からなかつたりしたところが、すらすら読めるはずです。第2部以降で学ぶことは、第1部で学んだマルチエージェント・シミュレーション技法の骨格に肉付けしていくようなものです。

授業の教科書としてこの本を使う場合には、授業時間の長さと履修学生の予備知識とを勘案して、いくつかの章を授業時間1コマの枠内でまとめて扱うことになります。私たちの経験では、講師も一緒にモデルを作る(パソコンのマウスやキーボードを操作する)と、授業の進度が速すぎて学生がついていけない事態を避けることができます。講師用のパソコンをプロジェクタで見せながら授業をすると、いっそう効果的でしょう。なお、必ず第1部を最初に、そして全ての章を教えて下さい。章の順序も、なるべく変えないで教えるのが望ましいと思います。他方、第2部以降は、授業で取り上げる章、自習に任せる章など、適宜アレンジして使って下さい。同じような技法が難易度の差で第2部と第3部の両方にまたがつて解説されている場合もありますが、まとめて教えてしまうのも一法でしょう。また、この本が心がけている独習者への配慮も、授業を進めるにあたつて、おそらく役立つものと信じています。

0.5 いよいよ始めるときです

独習であろうとも授業を通じての学習であろうとも、この本のモデルの作り方を真似て、そして課題をこなしていく中で、人工社会というまだなじみの少ないマルチエージェント・シミュレーションの発想と実際とに慣れていくことでしょう。そして、自分の関心ある社会現象をモデル化し、実際にマルチエージェント・シミュレーションを実行するときが、すぐ訪れるでしょう。

いずれにせよ、この本はマルチエージェント・シミュレータartisocのほとんど全ての機能を使いこなすための技法を盛り込んでいます。つまり、モデル作りに必要な素材、シミュレーション実行のさまざまな条件、そして経過と結果を見るようにする工夫はもれなく紹介します。あとは、自分の関心ある社会現象をモデル化するセンスを磨くことが大切になってきます。繰り返しますが、做って慣れることが肝腎です。

それでは、人工社会の世界に入っていって下さい。

第1章

なぜ人工社会を作るのか

1.0 人工社会はおもしろい

- 自律的な主体を互いに関係させ、社会全体のあり方を眺めます
- 全体のモデル化が難しくても局所的関係のモデル化で十分です
- 簡単な局所的関係が大域的様相を生み出す過程が見えてきます
- 人間社会だけに対象を限定する必要はありません
- 人工生命やネットワークもモデル化できます

1.1 人工社会の登場

人工社会(*artificial society*)という言葉は、人工知能(*artificial intelligence*)や人工生命(*artificial life*)などと並んで、コンピュータの発達とともに登場してきたもので、20世紀後半の学問・研究の変容の一面を表しています。つまり、知能や生命に関する研究とコンピュータとが結びついたのと同様、人工社会は社会に関する研究とコンピュータとが結びついたものです。そして、比較的新しい人工◎◎といった言葉の中でも、特に新しい言葉が人工社会です。学問の成果とも言える出版物に限れば、「人工社会」が初登場したのは、おそらくジョシュア・エプスタインとロバート・アクステルの *Growing Artificial Societies*(日本語訳書名は単に『人工社会』となっています)が刊行された1996年頃ではないでしょうか。

人間社会に関する学問の中には、コンピュータの発達に促されて発達した分野もあります。特に計量経済学は経済活動のモデル化や予測(シミュレーションの一種)で大きな成果を上げてきました。統計分析・数値処理の発達は社会学、政治学、教育学などさまざまな分野での実証研究の方法を大きく変えました。しかし、このような学問上の変化は、従来の人間観や社会観を変えるものではなく、既存のもの

の見方をコンピュータの発達が多様化・深化させたものと言えるでしょう。

これに対し、人工知能や人工生命という捉え方が知能とは何か、生命とは何かについて根本的な問い合わせをし、伝統的な見方の見直しを迫ったように、人工社会は社会に対するわたしたちの見方や理解を大きく変える可能性を孕んでいるのです。人工知能の研究は当初の期待を裏切る結果になったといわれていますが、知能とは何かという根元的な問いを追求していく上で大きな貢献をしたことは確かでしょう。同様に、人工社会の研究が将来どのような帰結になるかはともかく、人工社会の研究が人間社会の捉え方をさらに深めていくのに寄与してくれる信じています。

1.2 部分に注目しながら全体を捉える

人工社会という社会の新しい捉え方が従来のものとは異なる最大の特徴は、主体の相互作用に焦点を当てたボトムアップ・アプローチである、とよく言われます。

あえて比較対照的に考えると、従来の捉え方はトップダウン・アプローチと言えることになります。この意味は、人間の社会をモデル化するに際しては、社会全体の性質を取り込んだモデルを作る必要がある、ということです。だからこそ、たとえば数理モデルを解析的に解いたり数値的にシミュレーションしたりすることによって全体の性質をはっきりと描くことができるのです。

人工社会がボトムアップ・アプローチであるという特徴は、人工社会の研究では全体を表すモデルを作る必要がない、と言い換えるとその意味がはっきりするでしょう。つまり、主体間の相互作用にせよ主体と環境との相互作用にせよ、局所的な関係をモデル化しさえすれば、全体についての性質は自ずと現れる、という社会の捉え方です。社会全体の状態を誰かが決めてはいないという前提に立つ限り、社会全体の性質は自ずと現れる、というのはごく自然な発想です。

しかし、その自然な発想に基づく社会のモデル化が近年まで発達しませんでした。局所的相互作用のあり方を全体に結びつける具体的な手法を持ち合わせていなかったのです。しかし、コンピュータ利用法の発達が手法の発達を促しました。要するに、人工社会とは、局所的な相互作用についてモデルを作ると、コンピュータがシミュレーションを通じて大域的な状態を生み出してくれる、というアプローチなのです。

人工社会と似た言葉に人工市場があります。これは、人工社会の発想に基づいて、経済学が伝統的に扱ってきたシステムである市場とは異なるシステムを考え、それに相応しいモデルを構築して経済現象の研究をしようとしている研究アプローチです。人工市場の中の主体は、従来の経済学の主体であるホモ・エコノミクスとは異なる特徴を持っています。人工市場ほど研究の層は厚くありませんが、人工国会も考えられています。そこでは、一般の有権者、政治家(議員ないし立候補者)、団体としての政党は異なるタイプの主体として、異なる同種主体間あるいは異なるタイプの主体間で相互作用を繰り広げます。さらに人工国際社会の研究も始まっています。そこでは、多くの場合、国家やそれに準ずる政治勢力の相互作用が主たる分析対象です。

このように、人工社会を扱う上で基本となる「主体」について、人工社会では予め強い仮定を置いたりしません。主体の具体的なイメージとして、一人一人の人間を想定することも、企業や団体などの組織を想定することも、さらには国家を想定することも自由です。そして、異なるレベルの間の主体を関係づけることも可能です。人工社会の中で主体として想定されているものは、第一に個性(アイデンティティ、属性、役割など)を持っている、第二に周囲(の主体や環境)と相互作用できる、そして第三に相互作用を通じて、他の主体の個性に変化を引き起こしたり自分の個性が変化したりすることができる存在です。つまり、人工社会の基本的な考え方は、きわめて汎用性のある前提に基づいているのです。

1.3 基礎にあるマルチエージェント・シミュレーション

人工社会はコンピュータの発達が生み出した新しい社会の捉え方です。もっと正確に言えば、コンピュータを駆使したマルチエージェント・シミュレーション技法が基礎にあります。

もっとも、人工社会の研究は、マルチエージェント・シミュレーションという言葉もコンピュータも使わないと、1960年代末には既に行なわれていました。

それが「分居モデル」です。このモデルは、トマス・シェリングが1969年に発表した論文で提唱したものです。彼が注目したのは、地域社会が人種毎に分居する傾向にある現実が住民の排他意識とのような関係にあるのかという点でした。しかし、シェリングが用いた方法は住民の意識調査ではなく、サイコロを振りながらの机上の実験でした。シェリングの発想は、ミクロレベル(人々の個人的な好みや行動様式)

から推測できることが、必ずしもマクロレベル（社会全体の有り様）に反映するとは限らないことを定式化したもので、マルチエージェント・シミュレーションの発想そのものです。そして、彼の考案した分居モデルは、このことをはっきりと示す、単純ですが天才的なモデルです。マルチエージェント・シミュレーションを紹介する書物には、その草創期の典型例として必ず登場するモデルです。その意味で、マルチエージェント・シミュレーションのアイデアは、コンピュータとは独立に誕生したと言えましょう。

とはいっても、マルチエージェント・シミュレーションが発達したのは、コンピュータ技術の発達のおかげです。もともとコンピュータは、それを使う人間が何をやらせるのかを厳密に曖昧のない表現で指示してやる必要がありました。しかしコンピュータ技術の発達（オブジェクト指向とかエージェント指向とか呼ばれるプログラミング技術の開発）により、エージェントというプログラムがコンピュータの中やインターネットでつながった大きな世界の中で自律的に動き回り、他のエージェントや周囲の環境と相互作用するような仕組みを作り上げることが可能になりました。マルチエージェント・シミュレーションとは、この技術を利用したコンピュータ実験を指します。

そもそも「エージェント」という用語には、社会の構成員に通じるものがあることを感じさせます。実際、エージェントの持つ特徴として、自律性（外部からの指示どおりに行動するのではなく）、反応性（他のエージェントを含む周囲の環境のあり方に応じて行動を変える）、先見性（目的追求など率先して外部に働きかける）、そして社会性（他のエージェントと交信・相互作用ができる）などがあげられています。どれも社会科学で人間を主体として捉えるときに想定する特徴です。

したがって社会科学への導入が図られても決して意外ではありませんでした。かつては手間暇のかかった分居モデルのような実験も、今ではマルチエージェント・シミュレーション技法を用いると簡単にできるようになったのです。このような意味で、コンピュータに支えられたマルチエージェント・シミュレーション技法の発達が人工社会を生み出したと言えるでしょう。

この本でも、マルチエージェント・シミュレーションなし人工社会がどのようなものかをまず体験してもらうために、私たちが既に作った分居モデルをさっそく次章で実際に動かしてもらいます。サイコロを使ってシミュレーションを実行していた時代と、コンピュータに全てを任せられるようになった時代との違い、文字どおりの「隔世の感」を味わって頂きたいと思います。そして、前章でも述べたように、第1部の総括として、第

10章で分居モデルを実際に作ってもらいます。今までマルチエージェント・シミュレーションや人工社会は難しいとか面倒くさいとか思っていた人は、モデル作りが容易になった点でも「隔世の感」を味わうことになるでしょう。

1.4 人工生命も作れます(早く人工社会を作りたい人はとばしてかまいません。)

コンピュータを活用したマルチエージェント・シミュレーションの発達という観点から見ると、人工社会の近縁には、人工生命があります。これは、セルオートマトンの原理から出発したシミュレーションに基づきられた方法です。セルオートマトンというのは、もともと今日のコンピュータの基礎となる理屈と密接に関係しており、1940年代には原理が確立していました。しかしほどオートマトンが有名になるのは1960年代末以降のこと、数学者ジョン・コンウェイが考案した「ライフゲーム」がそのきっかけになりました。

ライフゲームは、セルオートマトンが敷き詰められた世界で、各セルオートマトンの状態(生か死)が周囲のセルオートマトンの状態に依存して変化するというものです。しかも、ライフゲームは、分居モデルと同様にコンピュータではなく、人の手と石ころそしてテーブルや床を使ったのだそうです。このきわめて単純なルールからできているライフゲームは、まもなくコンピュータでも再現されます。ライフゲームは人工生命に関する啓蒙書には必ず登場しますが、この本の第2部の総括を兼ねて、第20章で皆さんに実際に作ってもらいます。実に簡単に作れるのですが、見ていて飽きないモデルです。

やがて1980年代にはいると、ライフゲームという種子から人工生命と呼ばれるようになるマルチエージェント・シミュレーションの分野が発達します。ライフゲームは2次元セルオートマトンですが、それを1次元にする代わりにルールを少し複雑にしたもののがスティーブン・ウォルフラムの発明による、これまた有名なモデルです。そしてクリストファー・ラングトンは自己複製するセルオートマトンのモデルを実現させます。

要するに、ひとつひとつのセルオートマトンがマルチエージェント・シミュレーションのエージェントに対応しています。空間にエージェントを敷き詰めれば、それはすなわちセルオートマトンです。したがって、人工社会はオートマトンを用いた人工生命的のさまざまなモデルを再現することができるのです。

クレイグ・レイノルズの考案した「ボイド(人造鳥)・モデル」は、トリが群を作つて飛ぶ様子を少ないル

ルで再現したものです。このモデルは人工生命の例としても登場することがあります。セルオートマトンというよりは、人工社会の群行動モデルと言えるでしょう。オリジナルのボイド・モデルは3次元をCG化したのですが、第3部(第26章)で、2次元の簡易版を作ります。オリジナルのモデルのルールより簡単なルールによって、トリの群ができる、トリは障害物を避けていきます。

1.5 ネットワークも作れます(早く人工社会を作りたい人はとばしてかまいません。)

最近、ネットワークの面白さを強調する啓蒙書がいくつも評判になっています。ネットワークの理論は、もともとグラフ理論と呼ばれていた数学の一分野が近年急速に応用面も含めて発達したものです。人間社会のネットワークの研究としては、これまたコンピュータとは無関係に、1960年代にスタンリー・ミルグラムという社会心理学者が行なった郵便転送実験が再び脚光を浴びています。自分と世界中のひととは、平均して6人を仲介すると、全員が結びつく、という複雑なネットワークについての理論の出発点だからです。彼の他にも、アナトール・ラバポートとか社会科学の数理的・計量的研究に関心を持っている人なら誰もが知っている研究者がネットワークの分析を手がけています。ポール・エルデシュとアルフレッド・レーニーによるランダム・ネットワーク(ランダム・グラフ)の理論の登場もその頃でした。当時のネットワークの研究は、言うまでもなく、コンピュータ・シミュレーション技法と結びついていたわけではありませんでした。

しかし、1990年代に入って、ダンカン・ワツとスティーブン・ストロガツによるスマールワールド・ネットワーク、アルバート=ラズロ・バラバシとレカ・アルバートによるスケールフリー・ネットワークなどのモデルが提唱されると、こうした動きに刺激を受けて、さらにさまざまなモデルが提唱され、コンピュータ・シミュレーション技法の応用とともに、さまざまな分野で爆発的に流行しているようです。社会現象の研究分野も例外ではありません。

人工社会に組み込まれている発想や技法で、ネットワーク分析も可能です。そもそもセルオートマトンは格子グラフと見なすことができます。グラフ(ネットワーク)・セルオートマトン・人工社会は互いに似た構造を対象にしているのです。ネットワークの用語をマルチエージェント・シミュレーションの言葉に翻訳すると、頂点(ノード)がエージェントになり、枝(リンク)がエージェントどうしの何らかの意味での関係になります。

この本の第3部(第27章)では、ランダム・ネットワークが生成するモデルを実際に作ってもらいます。

第2章

マルチエージェント・シミュレータartisocの登場

2.0 artisocに初めて触れる

- 人工社会を作り、動かし、分析する道具artisocを紹介します
- artisocは使いやすくて本格的な人工社会を作れるシミュレータです
- artisocを通じて、人工社会とはどんなものかを体験します
- サンプル・モデルを実際に動かします

2.1 人間に優しいシミュレータの誕生

もっぱら理科系のツールとして開発されていたマルチエージェント・シミュレーションが、社会科学の方法のひとつとしても有望ではないかと言われはじめてから10年以上たちます。しかしこの手法がいまでもあまり浸透していないのは、社会科学の研究者や学生にとって、まずプログラミング言語から学ぶ必要があり、要するに敷居が高い手法だったことに起因すると思われます。この事情は、アプリケーションソフトが多数市販されている統計分析の手法と比べると一目瞭然です。

たしかにアメリカを中心に、人工社会を念頭においた汎用シミュレータはいくつか存在していますが、使いやすいシミュレータでは単純で定型的なモデルしか作れません。複雑でさまざまなモデルを作れる強力なシミュレータを操作するには高度な専門知識が必要です。やはり、使いやすiだけでなく、本格的な人工社会研究のできるツールが必要でした。

そのような状況で、数年前に登場したのがWindowsパソコン用のシミュレータKK-MASです。これは構造計画研究所が開発した、日本語環境のパソコンの上で、プログラミング言語やプログラミング技法を学ばないでも利用できる、しかも汎用性のあるマルチエージェント・シミュレータです。このKK-MASをプロトタイプにしつつ、社会現象のモデルを作り、シミュレーションを実行することを念頭に置いて開発された第2

世代のシミュレータがartisocです。つまり、artisocは当初から人工社会を作り分析するツールとして開発されました。artisocは、機能がKK-MASと比べて格段に充実しただけでなく、OSに依存しないので、パソコンの機種を選びません。その意味で、artisocはいっそう使いやすくなっています。易しさ(優しさ)と汎用性(強さ)を兼ね備えたシミュレータとして登場したのがartisocだと言えるでしょう。

artisocで複雑で高度なモデルを作れることは私たちが研究のために使用してきた過程で実証済みですが、この本では複雑なモデルは扱いません。artisocの入門教科書として、この本は使いやすさを強調していきます。

2.2 シミュレータやサンプル・モデルなどの取り込み

この本で使うartisocはartisoc textbookという版で、付録のCD-ROMに用意されています。これは市販されているartisoc academicと同じ機能を備えていますが、モデルの大きさやデータの取り込みなどに制限があります。この本では、artisoc textbookのことを単にartisocということにします。

それでは、付録のCD-ROMからartisocやサンプル・モデル、マニュアル、ヘルプなどを自分のパソコンに取り込みましょう。取り込み方は、Readmeja.txtに載っています。パソコンの機種により異なるので、自分のパソコンに合った方法で取り込んで下さい。この本は、基本的にはマニュアルやヘルプを参照する必要がないように書かれていますが、サンプル・モデルを改良したり、自分のアイデアをモデル化したりするような場合には適宜参照して下さい。

この本で、皆さんに作ってもらうモデルの例は、課題も含めて、山影研究室のウェブサイトにある教科書用のページ<http://citrus.e.u-tokyo.ac.jp/artisoc/textbook>で公開しています。もちろん、これが唯一の解答ではありませんが、思ったとおりにモデルができるときなどに参考にして下さい。

構造計画研究所の「MASコミュニティ」ウェブサイト<http://mas.kke.co.jp/>や山影研究室のウェブサイトには付録のCD-ROMには含まれていないサンプル・モデルがあります。適宜、ダウンロードして、さまざまな人工社会を体験して下さい。

なお、artisoc playerというソフトもあります。これはモデルの実行だけができるソフトです。構造計画研

究所のウェブサイトからインストールすることができます。サンプル・モデルは、同ウェブサイトの他、山影研究室のサイトにも載っています。どちらもダウンロード可能です。ただしartisoc playerで実行できるモデルは、xxxxx.modelではなくxxxxx.binaryというタイプのファイル形式になっているものだけなので、ダウンロードの際は注意して下さい。

2.3 artisocのモデルを動かす

いよいよartisocのモデルを動かして、人工社会を実感してみましょう。

artisocの起動は、一般のアプリケーションソフトの起動と同じです。アイコンをダブルクリックして下さい。

すると図2.1のようなウインドウが開きます。artisocの機能メニューが最上部にあります。そのすぐ左下には、AV機器の操作ボタンのようなものがあります。これは、シミュレーションの実行を制御するボタンです。さらにその下に、ツリーというウインドウが開いていて、そこに **Universe** と書き込まれています。これが、これから新しいモデルを作るためのスタンバイの状態です。

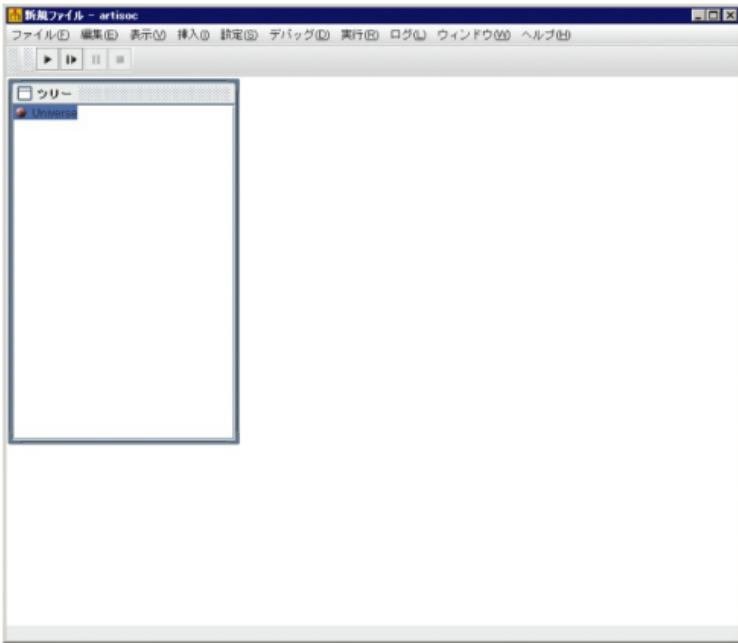


図2.1

新しいモデルの作成は次章以降に回して、ここでは、既にできあがったモデルを動かしてみることにしましょう。まず、山影研究室サイトの教科書ページから02materials.zipをダウンロードしてきて、展開(解凍、ダブルクリック)して下さい。02materialsという名前のフォルダがデスクトップにあるはずです。メニューの最も左に、**ファイル** ボタンがあります。そこにはプルダウン・メニューになっていますから、その中から**開く** を選んで下さい。そして02materialsフォルダの中にある02segregation.modelを開くように選んで下さい。このモデルは、シェリングの分居モデルの考え方を再現したモデルです。モデルが開くと図2.2のようなウィンドウになります。

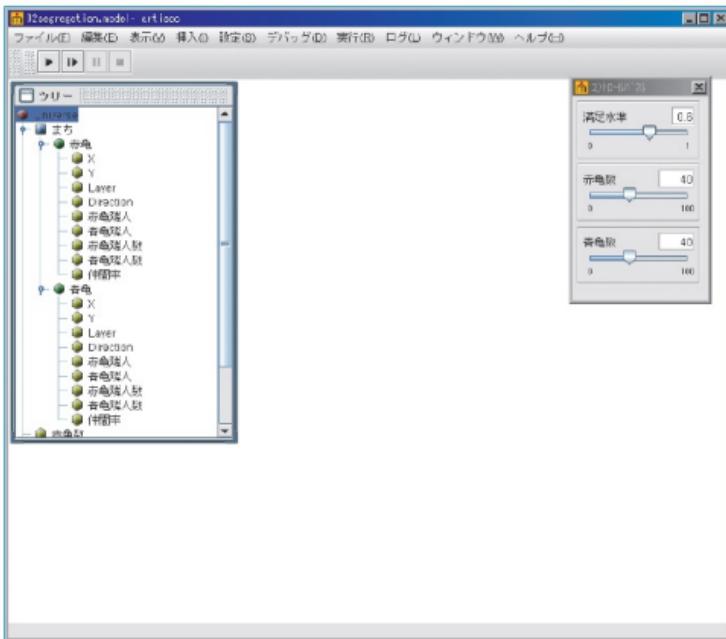


図2.2

とりあえず、実行ボタンを押して下さい。どれが実行ボタンだか分からない人は[図2.3](#)を参照して下さい。

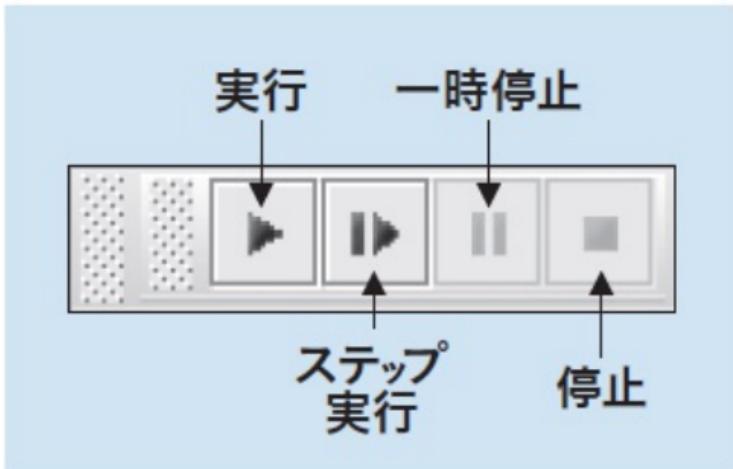


図2.3

これで、分居モデルのシミュレーションが始まりました。青い○や赤い○が動き回りやがて静止する画面（タイトルが「まち」）、折れ線グラフの画面（タイトルが「分居度の推移」）、数値の画面（タイトルが「変数値表示」）がウィンドウの中にあるはずです。これらは、いわば人工社会の地図と社会状況を表す指標です。画面が重なり合っている場合には、クリックした画面が最前面に出てきます。

停止ボタンと実行ボタンとを交互に押して、何回かシミュレーションを体験して下さい（[図2.4](#)参照）。

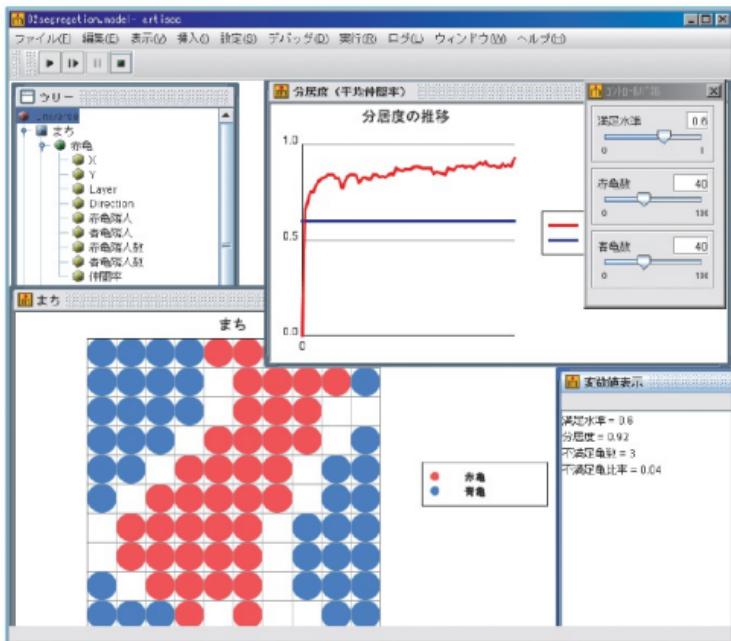


図2.4

以上の作業からわかるように、停止ボタンを押すとシミュレーションは終了し、実行ボタンを次に押すと、新しい初期状態から再びシミュレーションが始まります。この意味で、停止ボタンは、artisocではシミュレーションを終了させるボタンです。なお、一時停止ボタンを押すと、文字どおり、シミュレーション実行が一時停止(中断)し、再び実行ボタンを押すと、シミュレーションが継続します。

動かしてもらったモデルは、青い○や赤い○が「亀」ということになっています。周りに自分と同じ色の「亀」がある満足水準(閾値)より多くなるまで引っ越し続けるというルールになっています。最初は、ランダムに2種類の「亀」が住んでいる状態から出発し、だんだん分居が進行します。変化をゆっくり観察したいなら、(実行中なら一時停止ボタンを押してから)ステップ実行ボタンを押し続けて下さい。「まち」の様子だけでなくグラフにも注目して下さい。「満足水準」という水平線があります。これは個々の「亀」の満足水準(全

員共通)を表しています。「分居度」という折れ線は、各「亀」の周囲にどれだけ同類の人がいるのかを「まち」全体について表しています。シミュレーションを実行すると、分居が進み、やがて分居度は個々の「亀」の満足水準よりも高い状態になります。つまり、必要以上に「亀」は分居する結果になるのです。このモデルを最初に作ったシェリングに言わせると、これが一人一人の排他意識があまり強くなくても、地域社会はまるで相互排他的な人々が住んでいるように棲み分けされるメカニズムなのです。

2.4 シミュレーションの条件を操作する

ところで、パソコン・モニターの右の方に、コントロールパネルと書かれた図2.5のような小さなウィンドウが開いているはずです。(不完全なようなら、マウスでパネルの大きさを調整して下さい。)コントロールパネルは、モデルの中の設定をモデルの外から操作するためのものです。このモデルでは、満足水準とエージェント(「亀」)の数を変えられるようになっています。つまり、コントロールパネルのつまみを動かすことにより、条件をいろいろと変えてシミュレーションを実行することができるのです。





図2.5

まず、満足水準を0.4、赤亀数を40、青亀数を40にして、シミュレーションを何回か実行してみて下さい。折れ線グラフがフラットになったら停止ボタンを押しましょう。分居モデルの考案者のシェリングが言いたかったように、個々人の満足水準が0.4つまり近隣に自分とは異なる人が半数以上でも良いと考えているのにもかかわらず、社会全体としては0.7前後、つまり分居の程度がずっと高くなる傾向がある、ということがたしかに現れます。

それでは、適当に変えて、「亀」が引っ越ししあう様子を観察して下さい。ただし、赤亀数と青亀数の値の合計は必ず99以下（望ましいのは90以下）に設定して下さい。満足水準と実際の分居度とのギャップの大きさを実感して下さい。

以上が、人工社会というマルチエージェント・シミュレーションの基本です。研究のツールとしては、条件（「パラメータ」と言います）の違いが社会状態（分居モデルでは分居の程度）にどのような影響を及ぼすかを調べることになります。

2.5 中を覗いてみる(早く人工社会を作りたい人はとばしてかまいません。)

シミュレーションを実行してきた分居モデルの中身はどうなっているのでしょうか。ちょっと覗いてみましょう。「ツリー」ウィンドウの中の赤亀をハイライトして(クリックして **赤亀** の状態にして), メニューの **表示** から **ルールエディタ** を選んで下さい。ルールエディタが開きますが、そこには何やら文字がいろいろと書かれていることがわかります。これがエージェント(「赤亀」)を動かすルールです(図2.6)。

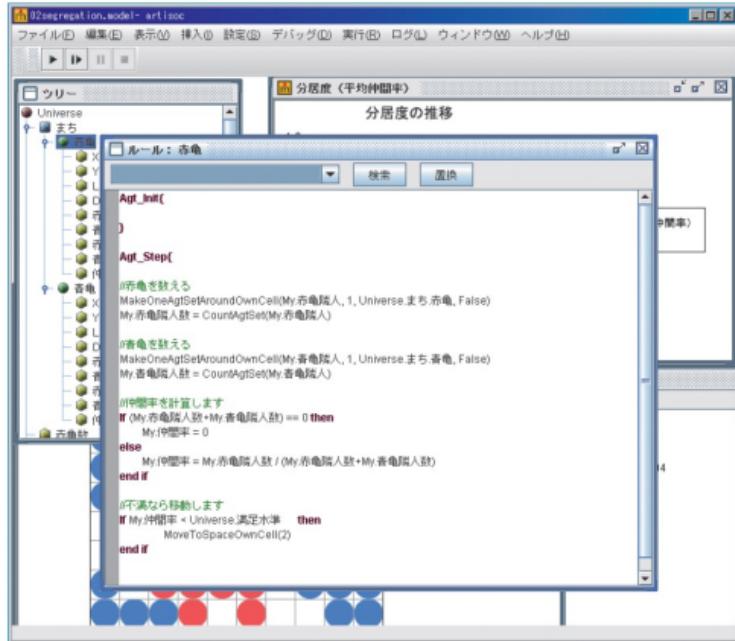


図2.6

ルールエディタの中は複雑そうに見えます。しかし心配しないで下さい。皆さんも第1部を修了するまでに、この程度のルールを自力で書けるようになっています。

次に、メニューの **設定** から、**出力設定** を選んで下さい。[図2.7](#)のような画面が開きます。

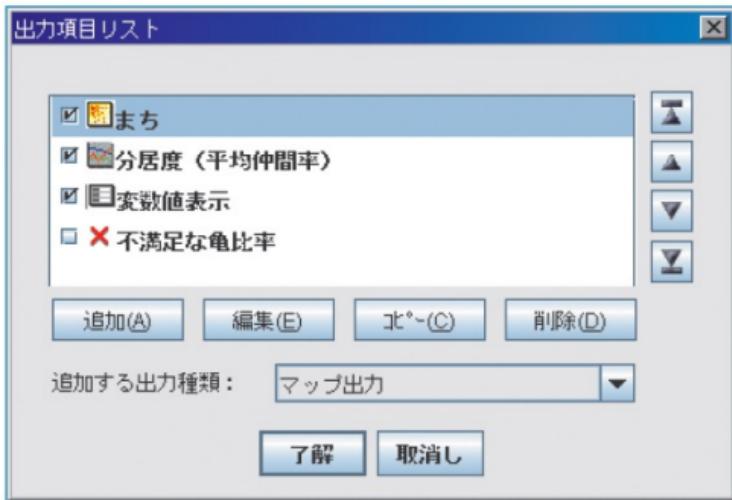


図2.7

ここで、シミュレーション実行の様子を「地図」と「折れ線グラフ」と「数値」で見せるように設定してあります。「棒グラフ」も設定されていますが、出力されないようにしてあります。

このように、ルールを書き込んだり、さまざまな設定をしたりして、モデルを作っていくことになります。

2.6 artisocを終了する

最後に、artisocを終了させましょう。

1. 停止ボタンを押して、シミュレーションの実行を終了させて下さい。
2. その状態で、artisocのウインドウを閉じる操作ボタンをクリックして下さい。
3. ファイル(今、実行した02segregation.modelのことです)を保存するかしないか尋ねるダイアログ・

ウィンドウが現れますから、「いいえ」(保存しない)をクリックして下さい。

これで、artisoc自体が終了しました。文書ファイルや集計ファイルを閉じても、アプリケーションソフトは終了しませんが、artisocではモデルのファイルを閉じる操作とアプリケーションソフトを終了する操作とは同じです。

2.7 自分で作りませんか

分居モデルを実際に動かしてみた感想はいかがでしたか。このようなモデルを自分で作ってみよう、というのがこの本の趣旨です。決して難しくはありません。それどころか、artisocを使って、ここで動かしてみた分居モデルよりもはるかに複雑なモデルも簡単に作ることができます。

次章から始まる第1部では、いよいよ白紙の新規ファイルの状態から、実際にマルチエージェント・シミュレーションを行なえるモデルを作っていきます。

artisocの出力設定には、デフォルトのマップ出力以外にも、グラフや値をそのまま出力することができる機能が備わっています。ここでは、分居モデルを例に、その簡単な紹介を行ないます。

マップ出力

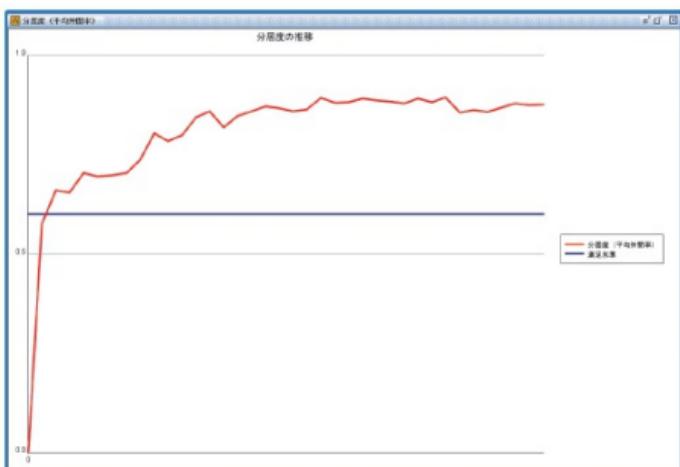
マップ出力では、XとYからなる二次元空間を表示することができます。また、XとYを空間以外のものとする(たとえば横軸Xと縦軸Yのグラフとみなす)ことも可能です。このような考えは、詳しくは第12章で学習します。



時系列グラフ

グラフには時系列グラフと、棒グラフの二種類があります。時系列グラフのX軸には最大表示ステップ数と目盛り間隔が設定でき、Y軸には目盛り間隔以外に、最大値と最小値が設定できます。そ

の名のとおり、ステップを刻むにしたがって、値の推移を動態的に観察することができます。第8章で登場します。



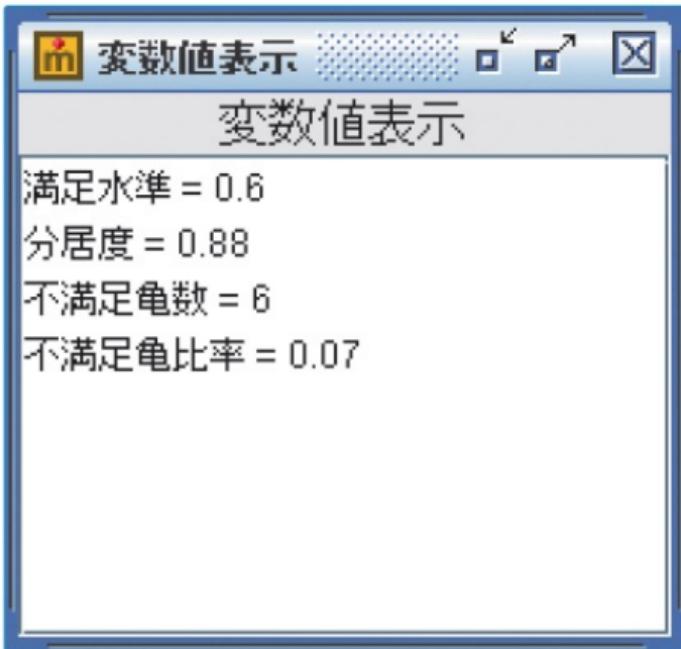
棒グラフ

artisocで表示できるもう一つのグラフは、棒グラフになります。たとえば、今現在の満足している亀と不満足な亀の数を比較したい時などに便利です。下の図は、分居モデルの棒グラフを表示させた例です。第30章で登場します。



値画面出力

値画面出力は、変数の値や、計算後の値を画面上に表示してくれます。分居モデルでは、満足水準、分居度、不満足な亀の数、不満足な亀の比率を毎ステップ計算して、表示させています。



ファイル出力

シミュレーションのログをファイルとして任意のフォルダへ出力してくれます。シミュレーションが終了した後の結果を分析するために、非常に便利な機能です。設定時には、ファイルを出力させたい場所、ファイル名、出力間隔(何ステップ毎に出力するのか)、そして区切り文字(タブやカンマ、スペース、あるいは任意の文字が指定できます)を選択します。第36章で学習します。

(保)

第1部

モデル作りの基本を身につける

ここでは、マルチエージェント・シミュレーションを実行するためのモデル作りの基本を学びます。作ったモデルを実際に動かし、実行中のシミュレーションを「見る」ための手法も学びます。第2章で紹介したシミュレータartisocを使います。artisocの基本構造に慣れてもらい、実質的なモデル作りの「基礎体力」をつけてもらうのが目的です。そのために、簡単なモデルをいくつも作ってもらいます。基本的な作業を繰り返すことで、「做うことで慣れる」ということを実践します。ここで学ぶ技法はどれも簡単ですが、それだけを用いて、シェリングが考案した(第2章で動かしてみた)「分居モデル」を完成させることができます。もっと洗練したモデルを作ることも可能です。

第3章

シミュレーションの準備をする

3.0 準備をすると後が楽です

- 鳥を飛ばすモデルを作り始めます
- モデルを組み立てるための設定方法を学びます
- シミュレーションを見るための設定方法を学びます

3.1 シミュレーションの3要素

シミュレーションを実際にやってみるには、少なくとも、次のようなプロセスが必要です。

1. シミュレーションするためのモデルを作る
2. モデルを実際に動かす(つまりシミュレーションの実行)
3. 実行の過程・結果を見る(つまりシミュレーションの出力)

モデルを作り、動かすということはシミュレーションとして当然ですが、最後の出力という要素も不可欠です。

それに対応して、次のような準備が必要です。

1. モデルの枠組み(土台)の設定
2. シミュレーションを実行させる環境の設定
3. 実行過程を見るための出力表示の設定

このような準備作業も含めて、自分でモデルを作り、マルチエージェント・シミュレーションを実際に行なってみるのは大変です。しかしこの本で使うマルチエージェント・シミュレータartisocは、モデルが簡単に作れ

るだけでなく、実行ボタンさえ押せば、シミュレーションが実行されます。さらに、シミュレーションの過程や結果の出力も簡単です。つまり、準備もモデル作りも実行もとても簡単です。この章と次章とで、以上の3要素を全て満たして、人工社会の初めてのモデルを完成させましょう。

さて、この章では、主に準備作業の手順を学びます。シミュレーションを実行させる環境の設定は、とりあえず、artisocが予め設定してある条件（デフォルトと言います）をそのまま使うので、わざわざ設定する必要はありません。したがって、

1. モデルの枠組み（土台）の設定
3. 実行過程を見るための出力表示の設定

を学びます。

3.2 artisocモデルの基本

artisocでは、人工社会のモデルは

- 「エージェント」と呼ばれる行動主体
- 個々のエージェントの属性（性質や役割）を表す「変数」
- エージェントが行動する（他のエージェントと関係する）「ルール」
- エージェントが行動（相互作用）する「空間」ないし「場」
- シミュレーションやモデル全体に関わる「変数」や「ルール」

を基本的な要素にしています。

そして、上のような基本要素を次のようにまとめています。

1. Universeと呼ばれる「全体」、「空間」、「エージェント（エージェント種）」という3階層からなる「ツリー画面」
 - ツリー画面を見れば、モデルの全体構造が分かります。

2. Universe, 空間, エージェント種の各々について指定できる「変数」
 - 変数といつても、それがとる値は数値だけではなく、文字列などさまざまな型をとることができます。
3. Universeとエージェント(エージェント種)にシミュレーションのルールを書き込むための「ルールエディタ」
 - artisocでは、複数のエージェントでも同じタイプならば、単一の「エージェント種」としてまとめてルールを指定します。
4. ルール
 - artisocのルールは広い概念で、エージェントの行動ルールだけでなく、シミュレーションを管理するルールや観察するための集計方法なども含みます。

artisocではさまざまなタイプの人工社会(マルチエージェント・シミュレーションのモデル)を作り、実行し、出力させることができます。モデル作りの基本は、Universeというまさに宇宙の中に、エージェントが相互作用する場を作り、相互作用のルールを指定することです。

3.3 モデルの枠組みを作る

この章と次章とで鳥が空を飛ぶモデル(「tobutori」)を作って、シミュレーションを実行し、その様子を観察します。この章では、鳥を飛ばすルールを書くまでの準備作業をします。

まず、実際にartisocを起動してみましょう(アイコンをダブルクリック)。起動すると、自動的に新しいモデルを作るためのスタンバイ状態になり、[図3.1](#)のような画面が出てきます。左上にある箱がツリー画面です。**Universe** が既に書き込まれています。

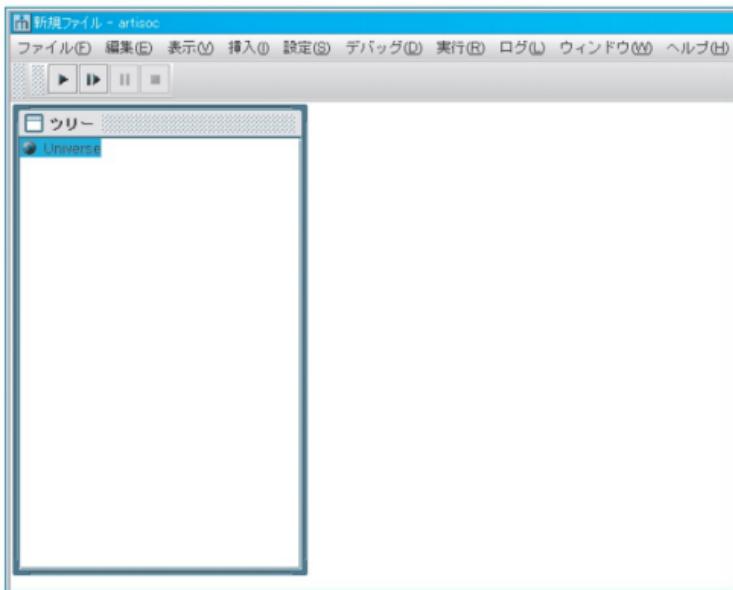


図3.1

これから、宇宙(Universe)に神様(創造主＝あなた自身です)が被創造物を作り出していくというイメージで、具体的なモデルを作っていきます。

新しい空間やエージェントを作っていくのは、「追加」という操作です。メニューの **挿入** にそろっています。

1. カーソルをUniverseに重ねてクリックして下さい。フォーカスされている状態(Universeだったら、 **Universe**)になります。この状態で、メニューから **挿入** をクリックし **空間の追加** を選ぶまでもOKです)。これをクリックして下さい。すると、空間を設定するための「空間プロパティ」ウインドウが開きます。空間の名称欄に、半角ローマ字でoozoraと書き込んで下さい。他は、そのまま(予め書き込まれている数字や選択されている選択肢のことをデフォルト値と呼びます)にしておいてください。

い(図3.2参照)。了解ボタンを押して下さい。これで、ツリーにはUniverseの下にoozoraという空間が設定できました。

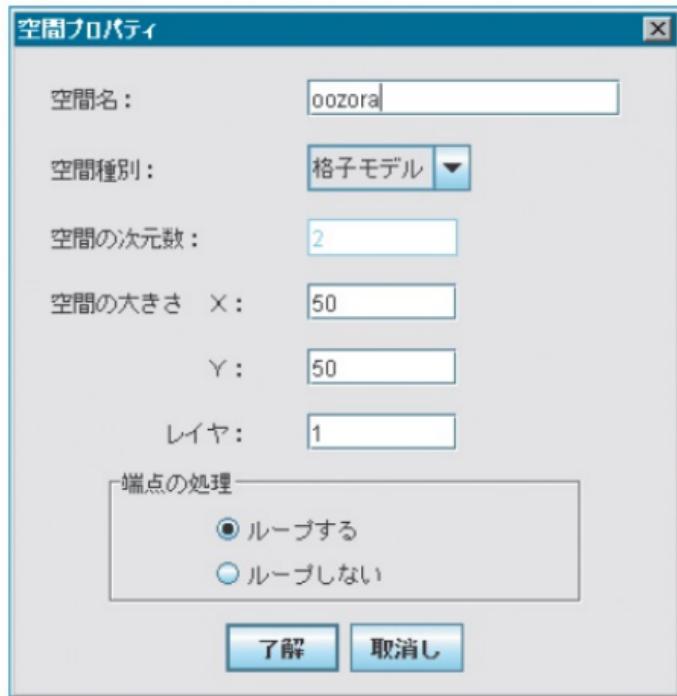


図3.2

2. **oozora** となっている状態で、メニューから **挿入** をクリックして、**エージェントの追加** をクリックして下さい。すると、空間を設定するための「エージェントプロパティ」ウィンドウが開きます。エージェント名の欄に、半角ローマ字で*tori*と書き込んで下さい。それから、エージェント数の欄に入っている0(デフォルト値)の変わりに、100と書き込んで下さい(図3.3参照)。そうしたら、**了解** ボタンを押して下さい。これで、ツリーにはUniverseの下のoozoraという空間に*tori*というエージェントが設定できまし

た。



図3.3

ここで注意して欲しいのは、toriという名前がついた100羽の鳥エージェントが、ツリーの中ではtoriという単一エージェント種にまとめられている点です。エージェントという言葉は、文脈により、複数のエージェントをまとめたエージェント種を意味したり、個々のエージェントを意味したりするので、気をつけて下さい。

3. ツリーの中のtoriのすぐ左の小さな○印をクリックして下さい。すると、toriの下に、ID, X, Y, Layer, Directionという文字が出てくるはずです。これは、エージェントを設定するとartisocが自動的に作ってくれるエージェントの5種類の属性変数です。たとえば、X, Yは各々、oozoraという空間(デフォルト値により、50×50に設定済み)のX座標とY座標を表しています(Directionについては第4章で学びます。IDやLayerの使い方については第3部で学ぶまでしばらく無視して下さい)。ここでは、100羽のtoriを設定しましたから、1羽毎にID, X, Y, Layer, Directionが設定されたことになります。これらは別の値をとることができます([図3.4](#))。

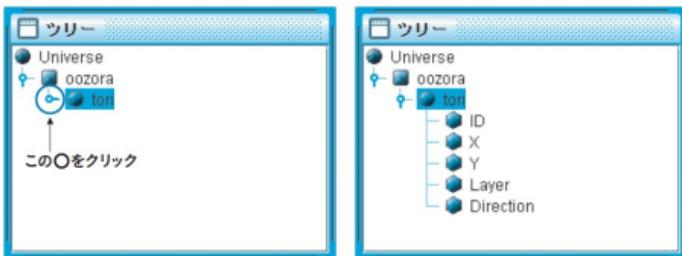


図3.4

4. ここまで作業が無駄にならないように、作業結果を保存しましょう。あるモデル（ここでは、「tobutori」）は、それについてのさまざまな設定を含めて、ひとつのファイルとなっています。この段階では、単に「新規ファイル」という名称になっているはずです。そこで、これに「tobutori」という名前を付けることにしましょう。まず **ファイル** メニューから **名前を付けて保存** を選択して下さい。tobutoriとファイル名を半角ローマ字で入力して下さい。正しく保存されていれば、artisocウインドウの最上部に「tobutori-artisoc」と書かれているはずです。

なお、OSの設定によっては表示されていないかも知れませんが、このファイル名には tobutori.modelというふうに拡張子「.model」が付け加えられます。これは、artisocでシミュレーションを実行するためのファイルであることを示しています。

せっかく作ったモデルが「なにかのひょうし」で消えてしまうと、とても悲しく（虚しく）なります。今後、ファイルは時々（特に実行ボタンを押す前に）「上書き保存」（**ファイル** の中の **上書き保存** をクリック）しておくことをお勧めします。

ところで、artisocは日本語対応のシミュレータです。したがって、oozora, tori, tobutoriといったローマ字表示の日本語は、大空、鳥とかトリ、飛ぶ鳥でも差し支えありません。しかし、この本でローマ字表記（半角英数字のフォント）を用いる理由は三つあります。ひとつは、次章以降学ぶようにルールエディタで書き込むルールが半角なので、全角にしたり半角にしたりすることから生じるエラーをできるだけなくすためです。二つめには、一部のパソコンOSで文字化けが生じる可能性があるからです。そして最後に、日本

語が混ざっているモデルのファイルを英語環境のartisocで実行するとエラーが生じる場合があるからです。したがって、以上の理由が障礙にならない人は漢字やカナ表記にしてもかまいませんが、慣れるまでは半角ローマ字の使用を勧めます。

3.4 シミュレーションの過程を見るための準備をする

次にシミュレーション実行を見るための設定をします。ここでは、空間oozoraを「マップ出力」して、toriがoozoraを飛ぶ様子を見ることができるようにしましょう。

1. **設定** メニューから **出力設定** を選んで下さい。すると、「出力項目リスト」ウィンドウが開きます。中央下の「追加する出力種類」が **マップ出力** になっていること（「マップ出力」がデフォルト値として設定されています）を確認したら、左隅の **追加** ボタンを押して下さい。すると、「マップ出力設定」ウィンドウが開きます。最上部の空間名にはoozoraと書かれているはずです。そこで、「マップ名」に oozora、「マップタイトル」にOOZORAと書き込んで下さい（[図3.5 参照](#)）。なお、「マップ名」はマップが出力されるウィンドウの名前で、「マップタイトル」はそのウィンドウにマップが表示されるときにマップのすぐ上に表記されるタイトルです。マップタイトルを空欄にしておくと、マップにタイトルが付きません（マップ名は必ず書き込んで下さい）。

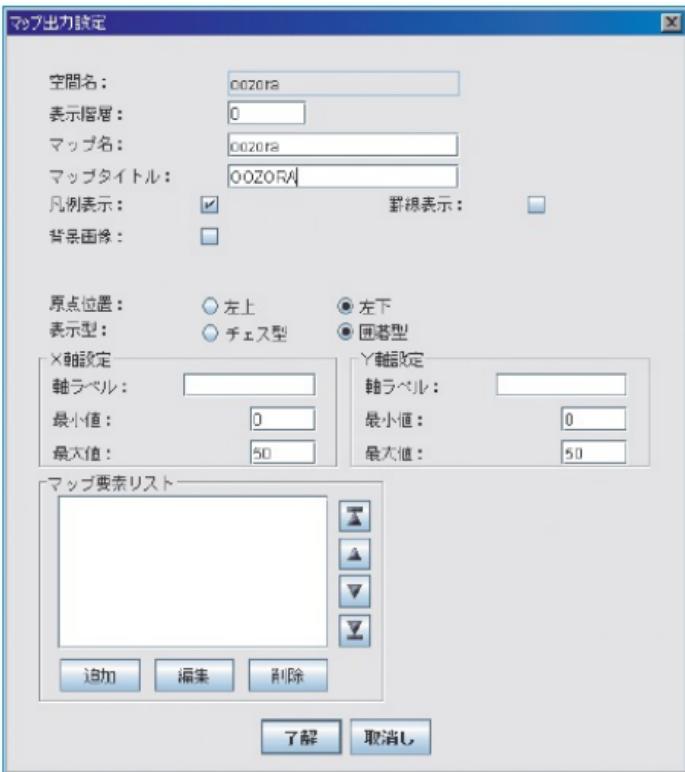


図3.5

2. 「マップ出力設定」の下部にある「マップ要素リスト」の **追加** ボタンをクリックして下さい。「要素設定」ウィンドウが開きます。出力対象はtoriとなっているはずです。「要素名」にTORIと書き込んで下さい(図3.6参照)。「要素名」は、マップ・ウィンドウの中に凡例として表示されます。

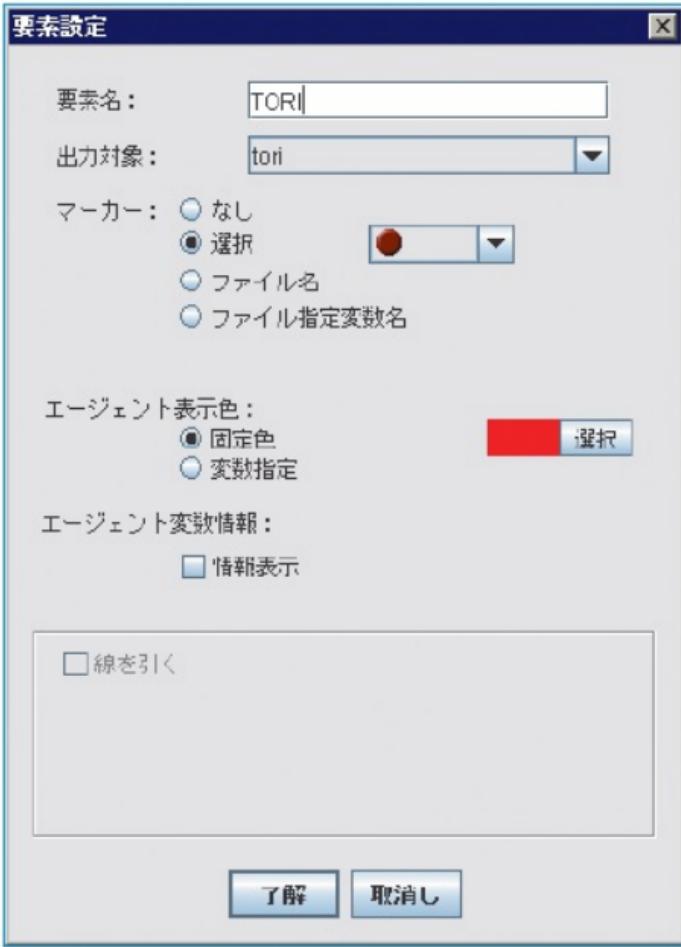


図3.6

3. 以上の作業をしたら(他は何もしないでデフォルト値のままにしておく),「要素設定」の **了解** ボタンを押し、「マップ出力設定」ウィンドウの「マップ要素リスト」にTORIが追加されているのを確認し,

「マップ出力設定」ウィンドウの **了解** ボタンを押して下さい。「出力項目リスト」にoozoraが追加されているのを確認したら、**了解** ボタンを押して下さい。

以上の作業で、パソコンのartisocウィンドウには、最初のツリー画面だけが残っているはずです。そろそろ上書き保存をして下さい。これから実行ボタンを押します。実行ボタンを押す前に、ファイルを保存するクセをつけた方がよいですよ。

3.5 作業が正しかったか確認する

これでモデル作りのための設定とシミュレーション実行を見るための出力設定は完了です。正しくできたかどうか、確認しましょう。

まず実行ボタンを押して下さい。[図3.7](#)のように、OOZORAとタイトルが書かれたマップ(その中には、凡例として、TORIがあるはずです)とコンソール画面(第8章で説明します)とが現れるはずです。

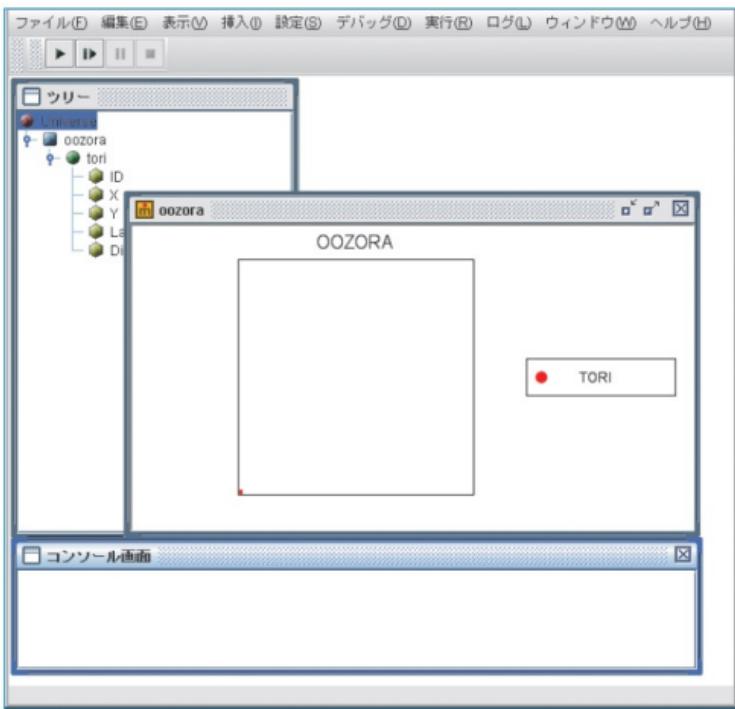


図3.7

マップの左下の隅に注目して下さい。丸いドットが見えるはずです(ドットの形と色は「要素設定」で指定しますが、artisocを起動してから以上の手順に従って作業すると、デフォルト値として●と赤色にあらかじめ設定されているはずです)。ここに100羽のtoriが重なり合って存在しているのです。ただ、toriがどのように羽ばたくのかについては、まだ何の作業もしていません。ただ、「存在している」だけです。

以上を確認したら、停止ボタンを押して下さい。

このままモデル作りの作業を続けたい人は、次章に進んで下さい。

ここでいったん作業を中断したい人は、artisocウィンドウを閉じて下さい。artisocが終了します。この際、ファイルを保存するかどうかを尋ねるダイアログボックスが開きます。**はい** を押すとこの時点での内容が保存され、**いいえ** を押すと最後に保存した内容のままになります。

新しく学んだ事項

- ツリーに空間とエージェントを作る（「挿入」メニューを使う）
- 空間を見るためにマップ出力設定をする
- 新規ファイルに名前を付けて保存する
- ファイルを上書き保存する
- 実行ボタン、停止ボタンを押す

第4章

エージェントを動かす

4.0 ルールが肝腎です

- 前章の続きとして、この章で鳥を飛ばすモデルを完成させましょう
- エージェントのルールエディタに行動ルールを書き込みます
- 動かすルールの初步を学びます
- 亂数を利用します
- 実際に動かして、シミュレーションの過程を観察しましょう
- ルールの変更や設定の変更に慣れましょう

4.1 エージェントの自律性とはどういうものか？

この章では、前章に続いて、鳥を飛ばすモデルを完成させて、鳥を飛ばせてみましょう。マルチエージェント・シミュレーションの最も基本的なところは、個々のエージェントが自律的に行動するという点です。toriというたくさんのエージェントの一體一体に、自律的に飛ぶという行動をとらせます。

ところで「自律的な行動」については、弱い（広義の）捉え方から強い（狭義の）捉え方までいくつかの段階があります。この章で完成させるモデルのエージェント（tori）は、もっとも弱い意味での自律的行動をします。そこで、まずもっとも弱い捉え方を説明しましょう。次章では、もっと強い捉え方の自律的行動を紹介します。

たとえば、100羽のtoriを動かしたいとき、エージェントに弱い意味での自律的行動をとらせるということは、1羽毎にどのように動くべきかを厳密に指示することをせず、行動の「ルール」を教えておけば、エージェントはそのルールに従って行動するということを意味しています。つまり、モデルを作る人間は、toriがどこに動くのかを事細かに指定するのではなく、動き方のルールさえ指定すればよいのです。toriというエージェント

は与えられたルールに従って、「勝手に」行動してくれます。このような意味で、エージェントには自律性があるということができます。

4.2 エージェントのルールエディタを開く

この章では、前章で準備した鳥が空を飛ぶモデル（「tobutori」）を完成させて、シミュレーションを実行し、その過程を見ることにします。

この章の最も大事なポイントは、エージェントの行動ルールを指定することです。artisocでは、エージェントの行動ルールは、エージェントの「ルールエディタ」に書き込みます。

artisocが起動していない場合には、artisocを起動させて、**ファイル**メニューの**開く**から、tobutori(tobutori.model)を選んで、開いて下さい。前章からそのまま続いてこの章に入った人には不要な作業です。以下では、tobutori-artisocウィンドウが開いている状態を前提にして、作業の説明をします。

「ツリー」ウィンドウにあるtoriを選択（クリック）して、**tori**にしてください。そしてメニューの中から**表示**を選択し、その中の**ルールエディタ**を選択して下さい。ルールエディタを開くことは、ツリーのtoriをダブルクリックしても（右クリックして**ルールエディタ**を選択しても）できます。[図4.1](#)のような「ルール：tori」というルールエディタの画面が開いたはずです。

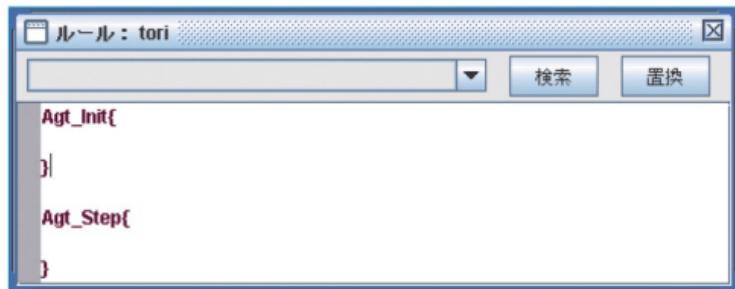


図4.1

ここで、`Agt_Init{ }`と`Agt_Step{ }`という2つの括りが、既にルールエディタの中に書き込まれていることに注目して下さい。`Agt_Init{ }`の括弧の間の部分にはシミュレーションの最初だけに実行するルールを書き込み、`Agt_Step{ }`の括弧の間の部分には、毎ステップ実行させるルールを書き込みます。シミュレーションの「最初」とか「毎ステップ」とかは後で詳しく説明します。

4.3 「動かす」ルールを書き込む

前章の最後で、実行ボタンを押すと、`tori`が左下隅に「存在」してはいましたが、全く動きませんでした。`tori`を動かすためには、`tori`の行動ルールが必要です。ここでは、次のような簡単な行動ルールを指定しましょう。

1. 全ての`tori`(前章で、100羽に設定しました)が`oozora`の中央部(X座標の値25,Y座標の値25:以降これを簡潔に(25, 25)と表記します)にいる。
2. 各`tori`は、勝手な(ランダムな)方向を向いている。
3. 各`tori`は、毎ステップ、「1」ずつ、その方角に飛び続ける。

`artisoc`では、座標系は通常の数学での用法と同じになっています。左下隅が原点(0, 0)で、右上方に進むほど、X座標もY座標も大きくなります。また、角度は基準点から右水平方向を0度とし、左回り(反時計回り)に角度が増えます。真上が90度、真下が270度です。

ここで「毎ステップ」という言葉が出てきました。ステップとは、`artisoc`の時刻単位の呼び名です。時刻や時間は、ふつう、日・時・分・秒という単位で測ります。このような「流れる時間」の測り方に対して、コンピュータのなかで実行するマルチエージェント・シミュレーションでは、全てのエージェントがルールにしたがって行動することを1時点として時刻を刻みます。この単位を「ステップ」と呼びます。全エージェントのルールを実行するにはある程度の時間が必要ですが、その時間の長さは無視します。また、`artisoc`の中でのある時点と次の時点との間(ステップ間隔)が現実の社会でどれだけの時間に対応するかは、別に考えます。

最初に1回だけ実行させるのは、上の行動ルールの1と2です。これを、artisocのルールエディタの中のAgt_Init{}にルールとして指定しましょう。

まず、Agt_Init{} の次の行に、My.と半角英数字で書き込んで下さい。その際、必ず、半角英数字を使って下さい。

すると、どうでしょう。M, yそして「.」を入力したとたん、エージェントの変数が縦に並んだメニューが現れたはずです。これは、モデルを作る上でartisocが用意した「支援機能」で、その中から適当な変数を選択すればよいのです。この支援があるおかげで、入力する手間が省けるだけでなく、ミスタイプによるエラーが起らなくなります。この段階で現れるID, X, Y, Layer, Directionの5種類の変数はartisocが自動的に設定した変数ですが、しばらくX, Y, Directionの3つだけを使います。

最終的には、下のようなルールを書き込みます。文字も記号も全て半角で入力して下さい。ルールエディタの中では、自分で定義した空間名、エージェント名、変数名以外の文字や記号は、半角で書き込む必要があります。既に指摘したように、artisocのルール表記に慣れるまでは、空間名、エージェント名、変数名も半角ローマ字で表すことを勧めます。なお、ローマ字の大文字と小文字は区別されません。たとえば、my.X, MY.X, My.Xは全て同じと見なされます。したがって、視覚に訴え、入力ミスに気づきやすいように、適当に大文字と小文字とを使い分けてかまいません。

```
Agt_Init {  
    My.X = 25  
    My.Y = 25  
    My.Direction = Rnd() * 360  
}
```

何となくわかるかも知れませんが、以下の5点に注意を要します。

1. My.XとかMy.Yなど、エージェントの変数(X, Y, Directionなど)の頭に「My.」を付けた変数は、たくさんある(このモデルでは100体の)エージェントの1体毎に指定される自分(my)だけの変数を表し、エージェント毎に異なる値をとることができます。
2. Directionとは、移動方向を指定する特別の変数です。空間(oozora)上で、エージェントのいる地

点から3時方向(右水平方向)を0度とし、反時計回りに1回転を360度とする値をとります。

3. Rnd()とは、0から1の間の一様乱数を発生させる関数です。「Rnd()」そのものが0以上1未満のランダムな値になります。かっこの中には何も書き込みません。また、Rndの「d」と「()」の間に空白(スペース)は入れないで下さい。
4. 「*」はかけ算(\times)を表します。したがって、Rnd() * 360というのは、0以上360未満の値になります。
5. 全ての式について、等号(=)は、その右の値を左の変数の値にしまう(代入する)という操作を意味します。等号の右側と左側が等しい、と比べているわけではありません。数学の記号とは意味が違うので気をつけて下さい。

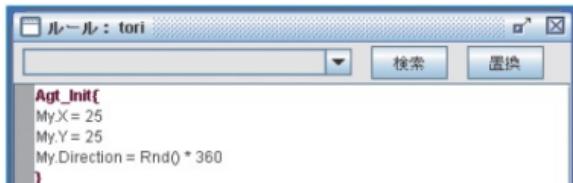
次に、毎ステップ実行するルールを指定します。それは、最初に指定された飛び立つ方向(My.Direction)に、「1」ずつ飛んでいく、というものです。これは、Agt_Step{}部分に

```
Agt_Step{  
    Forward(1)  
}
```

と書きます。Forward()とはかっこ内の数値だけ、そのエージェントが向いている方向に、前進する(forward)というルールです。ここで注意を要するのは、「1」というステップ毎の飛翔距離は、oozoraという空間を50×50に設定したことに対応する「1」です。つまり、左下端から真横または真上に飛び続けると、50ステップ後に右端または上端に着くという意味です。

結局、「ルール:tori」というルールエディタには、[図4.2](#)上図のようにルールが書き込まれます。

これでtoriエージェントの行動ルール指定は完了です。この辺で、「上書き保存」([ファイル](#)から[上書き保存](#))しておきましょう。もうtobutoriのシミュレーションを実行できます。



```
Agt_Step{  
    Forward(1)  
}
```

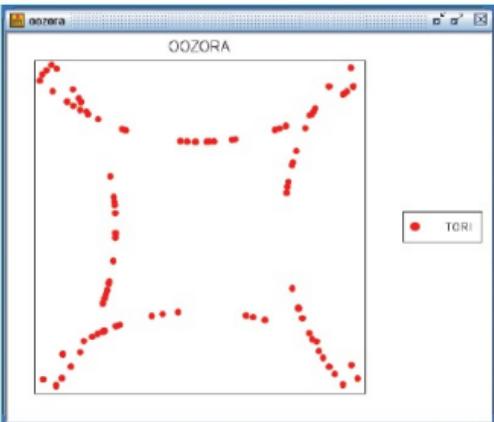
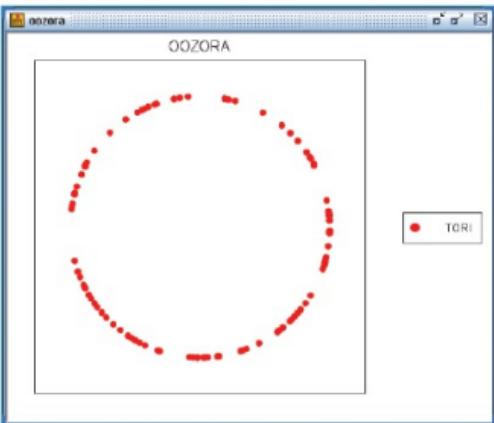


図4.2

4.4 いよいよ実行

では、実行ボタンを押して下さい。100羽のtoriが中央部から四方八方に飛び去るのが、OOZORAの中に見えるはずです(図4.2中図参照)。

適当なところで、停止ボタンを押して下さい。何回か、シミュレーションの実行と停止(終了)を繰り返しましょう。

パソコンの性能が良いと、シミュレーションの実行が速すぎて、マップ上のエージェントの動きが単に速いだけでなく、ぎこちなく見えることがあります。そのような場合は、わざと実行を遅くすると動きが緩慢になります。そのためには、メニューの「設定」から「実行環境設定」を開き、「実行ウェイト」を大きく(たとえば100に)します。

さて、シミュレーションが始まつてしまらく経つと、図4.2下図のようにtoriは上下左右の縁で反転しているように見えますが、そうではありません。oozoraという空間は、ループしているのです(「空間プロパティ」のデフォルト値です)。空間がループしているということは、上端と下端、左端と右端とがつながっているということです。したがって、エージェントが左から右に移動して右端に着くと、すぐに左端から再登場するのです。このようにループしている平面は、ドーナツの表面と同じで、専門的にはトーラスという2次元曲面です。ループしている空間(X-Y 2次元平面)は、ちょうど図4.3のように、ドーナツの表面を縦と横に開いたものだと考えて下さい。

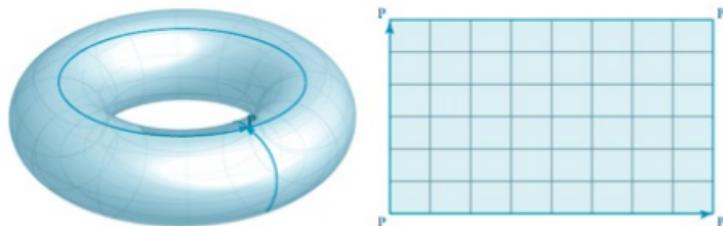


図4.3

4.5 動かすルールや設定変更に慣れる

このようにして完成したtobutoriモデルをいろいろと修正してみましょう。

toriの数を100羽から1000羽に増やす。

1. 「ツリー」の中のtoriを選択する
2. メニューの 表示 から プロパティ を選択する
3. 「エージェントプロパティ」画面が開くので、その中のエージェント数を1000にする

これで完了。それでは、tobutori(B)という名前を付けて保存してから、実行ボタンを押して下さい。既に作ったモデル(ファイル)に新しく別の名前を付けて保存するには、ファイル から 名前を付けて保存 を選んで作業します。

toriを左上端から飛び立たせる。

1. 「ツリー」の中のtoriを選択する
2. メニューの 表示 から ルールエディタ を選択する
3. My.X, My.Yの代入式の値を25, 25から0, 50に変える

これで完了。それでは、tobutori(C)という名前を付けて保存してから、実行ボタンを押して下さい。

toriは飛ぶ方向を毎ステップでたらめに選ぶ。

1. 「ツリー」の中のtoriを選択する
2. メニューの 表示 から ルールエディタ を選択する
3. Agt_Init{ }内のMy.Direction = Rnd() * 360という代入文を下のAgt_Step{ }内に移す(カット・アンド・ペースト機能を使うと便利です)。

これで完了。それでは、tobutori(D)という名前を付けて保存してから、実行ボタンを押して下さい。

このように、簡単に修正できます。そして、各々、*tori*の飛び方にもいろいろな違いが現れたはずです。

新しく学んだ事項

- エージェントのルールエディタにルールを書く
- My.変数の入力支援機能
- Agt_Init{ }とAgt_Step{ }の違い
- My.につづくエージェントの変数X, Y, Directionの使い方
- 右辺の値を左辺の変数に代入する代入文(=)
- かけ算の記号 *
- Rnd()
- Forward()
- 空間は(デフォルト状態だと)ループしている
- ルールエディタの中で、ローマ字の大文字と小文字の区別は見やすいように
- シミュレーションの実行をわざと遅くする(「実行ウェイト」の変更)
- 別の名前をつけて保存する

練習問題

この章で学んだことの復習を兼ねて、練習問題を試してみましょう。

4.1

中央下を出発点にして、上半分のさまざまな方向に*tori*を飛ばす。



My.Direction = Rnd() * 180

oozoraの中央から毎回ランダムなスピードでtoriを飛ばす。



Forward(Rnd())

第5章

エージェントに判断させる

5.0 行動の選択が自律性の根本です

- 鳥の飛び方をいろいろと変えたモデルを作りましょう
- 状況に応じて異なる行動をさせる基本は「場合分け」です
- artisocでの「場合分け」のルール表記「イフ文」を学びます
- 「場合分け」のいくつかの技法を身につけましょう
- 亂数のいろいろな利用法を学びます

5.1 本格的な「自律的な行動」とは何か

前章ではエージェントが勝手に行動するという意味での自律性をモデル化しました。この章では、もう少し、意味の強い自律性を持ったエージェントをモデル化します。

例を用いて、強い捉え方の自律的行動を説明しましょう。今、P地点にいるエージェントがQ地点に移らなければいけないとしましょう。P地点からQ地点に行くには、J, K, Lの3通りの方法があります。移動費用ではJ, K, Lの順で高額なのですが、移動時間はJ, K, Lの順で短時間です。他方、風景の良い順では、K, J, Lです。エージェントが急いでいれば、高額でも最速のJを選ぶでしょう。もし時間に余裕があればKを選び、懐が寂しければLを選ぶかも知れません。この例では、エージェントにとっての望ましさの基準に従って複数の選択肢から特定のものが選ばれます。

このように、強い捉え方での自律的行動では、エージェントが複数の可能性から一つを選択する、いうことが前提になります。言い換えると、エージェントに自律的行動をさせるには、モデルを作る側は、選択肢（行動のレパートリー）と、どのような場合にどの選択肢が選択されるのか、といったルールをはっきりさせておく必要があります。

あらかじめ、さまざまな可能性を設定して、ある場合にはどれが選択されるのかを指定することを「場合分け」または「条件分岐」といいます。この章では、「場合分け」をartisocのルールとして、エージェントのルールエディタにどのように書き込むかを学びます。

5.2 エージェントに何をさせたいのかを図ではっきりと表す

(プログラミングに慣れている人はとばしてもかまいません。)

モデルを作る作業にはさまざまな段階がありますが、エージェントの行動ルールを明記することは基本中の基本です。この、行動ルールの明記には、2つの段階があります。ひとつは、まず、私たちがエージェントに何をさせたいかをはっきりさせることです。もうひとつは、それをシミュレーション用のモデルの一部としてはっきりさせることです。つまり、artisocのルールエディタに正しく書き込むことです。

実は前章でも、既にこの2段階を実践しています。

- 第1段階(ルールの決定): toriの初期状態(位置と向き)や飛び方を決める
- 第2段階(シミュレーション・モデルでの表現): artisocのルールエディタの中にルールを書き込む

とても単純なモデルであるtobutoriの場合、第2段階の作業はもちろん必要ですが、第1段階については、自分の頭のなかで、エージェントに「こういうことをさせる」内容がはっきりしていれば、必ずしも、メモを書いたり、絵を描いてみる必要はありませんでした。

これから学ぶ「場合分け」は、ルールが複雑になります。第1段階の作業を、頭のなかだけでなく、紙に図示したりする習慣をつけましょう。図示する方法のひとつに、「フローチャート(流れ図)」があります。これは、もともとコンピュータに何かをさせようとするときに、その作業の「流れ(フロー)」を「図(チャート)」に描いたことに由来します。フローチャートを描くのに際しては、「場合分け」を見やすく描く方法が標準化されています。それは、コンピュータにさせたいことをコンピュータに分かる方法で指定する(プログラミング)ときに、「場合分け」がとても重要で、しかも頻繁にあるからです。artisocでルールエディタにルールを書き込むことはプログラミングではありませんが、ルールに「場合分け」を用いることが多いので、フローチャートを書い

ておくとルールを書き込むときに便利です。

単純ですが、tobutoriの例では、フローチャートは図5.1のようになります。

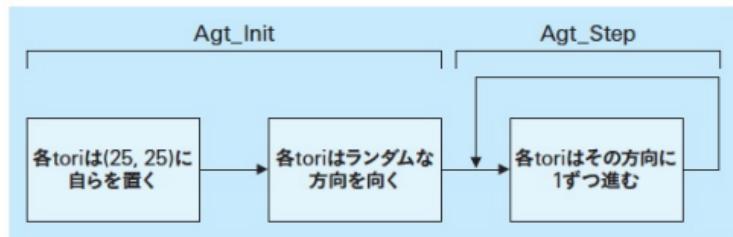


図5.1

5.3 「場合分け」を図示して、正しく理解する

「場合分け」の基本は、当たり前ですが、どのような場合があるのか、そして場合毎にエージェントは何をするのかをはっきりさせることです。ここでは、あまり「自律的行動」らしくありませんが、「場合分け」のルール化の基礎を学びます。

toriが中央の下から上に飛び上がり、30ステップ以降は各toriは勝手な方向に飛ぶ。

具体的には、次のようにエージェントを行動させてみましょう。

1. oozoraの下端中央(25, 0)にいる
2. そこで真上を向く
3. 30ステップになっていなければ、毎ステップ、「1」ずつ、前方に飛び続ける
4. 30ステップ目に、飛んでいく方向を勝手に決める
5. 30ステップを超したら、毎ステップ「1」ずつ、前方に飛び続ける

ここで、3, 4, 5が「場合分け」です。30ステップ「より少ない場合」、「ちょうどの場合」、「より多い場合」で、各々の場合に、行動ルールが異なっています。ではこの行動ルールをフローチャート化しましょう。

[図5.2](#)のフローチャートを眺めてみると、「30ステップより少ない場合」と「より多い場合」とが、同じ行動ルールになっていることに気づくはずです。ですから、実は、スマートなフローチャートは[図5.3](#)のようになります（もちろん、[図5.2](#)でもまちがいではありません）。

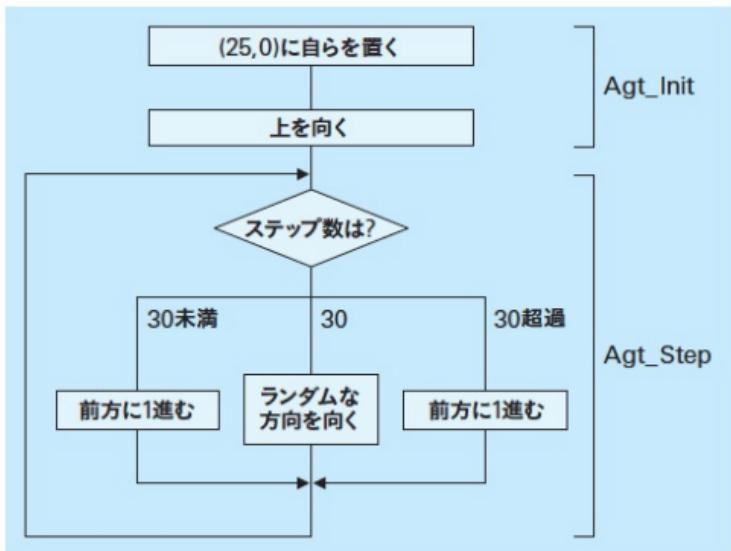


図5.2

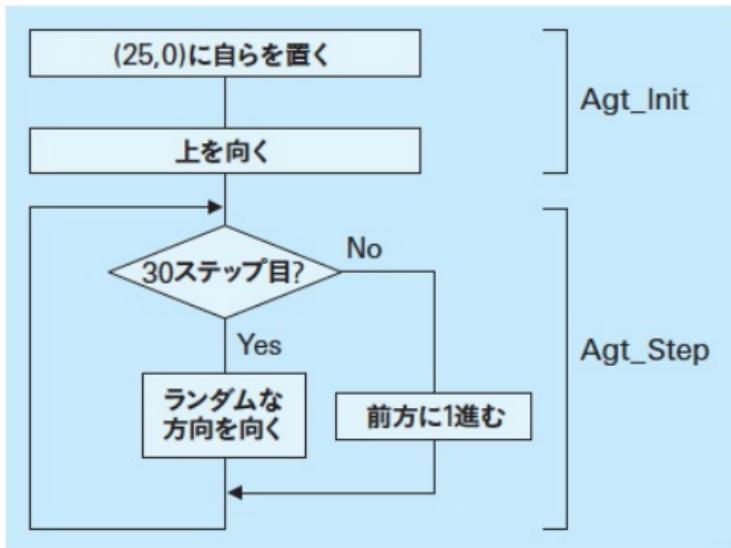


図5.3

5.4 「場合分け」ルールでエージェントに判断させる

まず、モデル作りの準備作業をしましょう。第3章(3.3と3.4)でtobutoriモデル作りの準備をしたのと、全く同じ作業をして下さい。なお、「マップ出力設定」の「要素設定」のウィンドウで「エージェント表示色」が第3章のとき(赤色)と違う(おそらく青色)かもしれません。これは、artisocを終了しないで作業を続けると、自動的に固定色の色を変えてくれるためです。マーカー(エージェントの形、●■▲✿など)やエージェントの表示色(固定色では)は適当に選択してかまいません。

この新しいモデルは、hanabiと名前を付けて保存して下さい。なぜhanabiという名前のモデルなのかすぐにわかります。

では、スマート版フローチャートにしたがって、いよいよルールを書き込みましょう。toriのルールエディタを開いて下さい。Agt_Step{ }には、「場合分け」した行動ルールを次のように書き込みます。新しい表現が

いくつか登場しますが、すぐ後で、説明します。

```
Agt_Init{  
My.X = 25  
My.Y = 0  
My.Direction = 90  
}  
Agt_Step{  
If GetCountStep() == 30 Then  
    My.Direction = Rnd() * 360  
Else  
    Forward(1)  
End if  
}
```

以上を書き込んだら、上書き保存をしてから、とりあえず、実行してみましょう(図5.4)。なぜ、このモデルをhanabiと名付けたのかわかりますね。

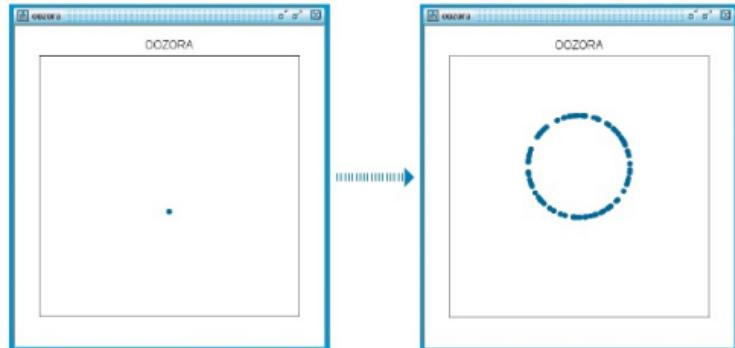


図5.4

では、上のルールを説明します。Agt_Init{ }の部分のルールは、既に前章で学んだものですから説明を省略して、Agt_Step{ }の部分について説明しましょう。

1. まず、大きな構造である

```
If XXXXX Then  
    YYYYYY  
Else  
    ZZZZZZ  
End if
```

に注目して下さい。これは「もし(If) XXXXX ならば(Then) YYYYYYを実行し、そうでなければ(Else) ZZZZZZを実行して、完了(End if)する」という「場合分け」の基本構造です(英語の文章に似ています。コンピュータに関する技術が、アメリカで発展してきた名残です)。この、Ifで始まり、End ifで終わるルール表記を「イフ文」と呼ぶことがあります(理由は明白ですね)。

- ここで、イフ文のXXXXXの部分にあるGetCountStep() == 30が新しく登場した表現です。GetCountStep()はartisocがシミュレーションを実行しはじめてからのステップ数です。GetCountStep() == 30はステップ数がちょうど30であることを意味しています。数学の等号(=)を2つ並べた記号は、記号の左辺の値と右辺の値が同じであることを表しています。つまり、右辺の値と左辺の値を比較して等しいことを表現しているのです。この記号は、通常、条件の中(イフ文のXXXXXのところ)でしか使われません。代入(=)と等しい関係(==)とを混同しがちなので、気をつけて下さい。

したがって、上のルールは「もしステップ数がちょうど30なら、飛ぶ方角をランダムに選びなさい。そうでなければ、決まっている方角に毎ステップ1だけ飛び続けなさい。これで場合分けは完了」となります。このルールがスマート版フローチャートに対応していることを確認して下さい。

5.5 「場合分け」のいろいろな技法

これから、完成したhanabiモデルをいろいろと修正してみましょう。その過程で、新しい表現方法も学びます。

30ステップ目、toriは勝手な方向を向くのではなく、左か右の水平から上方30度の方角(つまり0度

から30度の間または150度から180度の間)を向く。あとは同じルールで飛ぶ

まず全体のフローチャートを書いて下さい(図5.5)。

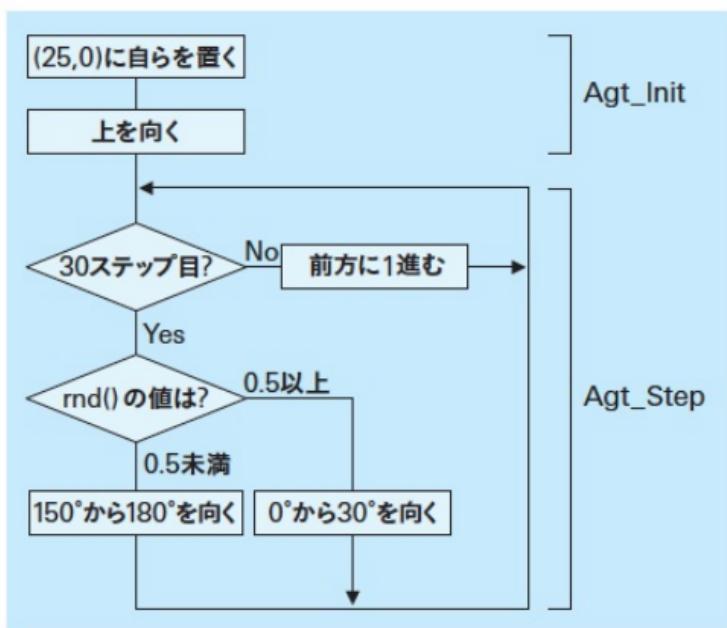


図5.5

フローチャートをよく見ると、場合分けが「入れ子」状態(2重)になっているのに気がつくでしょう。artisocのルールでは、イフ文を「入れ子」構造にすればよいわけです。

```
Agt_Step{  
If GetCountStep() == 30 Then  
  If Rnd() < 0.5 Then  
    My.Direction = 180 - Rnd() * 30  
  Else  
    My.Direction = Rnd() * 30
```

```
End if  
Else  
    Forward(1)  
End if  
}
```

ここで「入れ子」の内側のイフ文に注目して下さい。IfとThenの間の条件が、 $\text{Rnd}() < 0.5$ となっています。「<」は左辺と右辺との大小関係を比較して、左辺の値が右辺の値より小さい（右辺の値が左辺の値より大きい）ことを表しています。つまり、このイフ文全体は、一様乱数を使って、だいたい半分（0以上0.5未満、0.5以上1未満）にtoriエージェントを分ける技法です。厳密に半々にする必要がなければ、エージェント数にかかわらず2等分する便利な技法です。

それでは、hanabi(B)という名前を付けて保存してから、実行してみましょう（図5.6）。

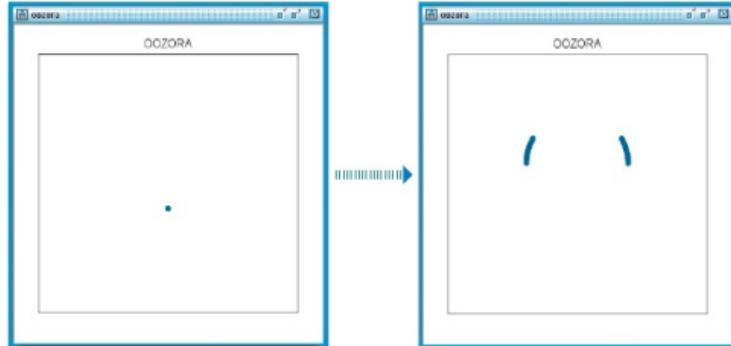


図5.6

30ステップ目では左右に分かれ、さらに40ステップ目以降では、毎ステップ左右10度の範囲でランダムに方向を変える

なお、今までには、向きを変える30ステップ目では前方に飛びませんでしたが、ルールを簡単にするために、30ステップ目も40ステップ目も前方に飛ぶようにしましょう。

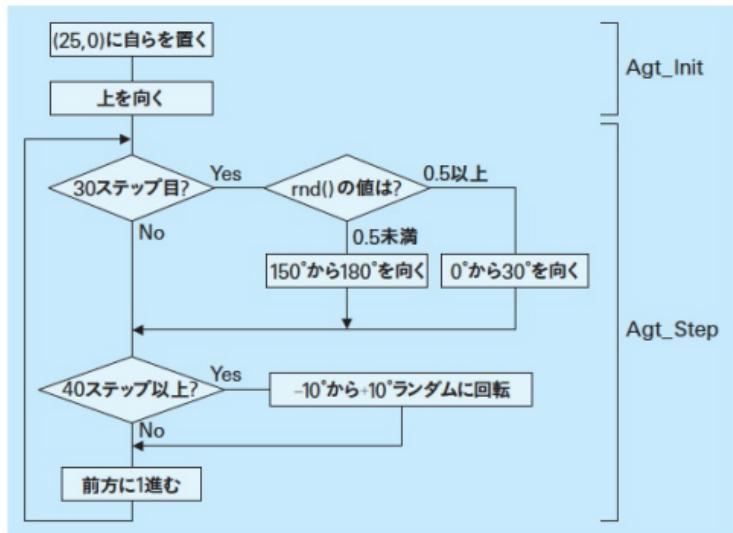


図5.7

```

Agt Step{
If GetCountStep() == 30 Then
  If Rnd() < 0.5 Then
    My.Direction = 180
  Else
    My.Direction = 0
  End if
End if
If GetCountStep() >= 40 Then
  Turn(Rnd() * 20 - 10)
End if
Forward(1)
}

```

ここで、新しいルール表現が2つ登場しました。

1. ひとつはIf XXXXX Then YYYYY Else ZZZZZ End if ではなく、If XXXXX Then YYYYY End if という表現です。このElse ZZZZZがないものは、イフ文の簡略版です。単に、条件 XXXXXに当てはまるときにYYYYYを行なえ、という意味です。なお、条件部分に出てきた「>=」

は、左辺の値が右辺の値より大きいか等しいことを表しています。

- もうひとつはTurn(Rnd() * 20 - 10)です。Turn(D)とは、自分が向いている方向を基準にして左回り(反時計回り)にD度だけ方向を変えるというルール表現です。

なお、上のTurn(D)のDに注目して下さい。Rnd() * 20は0以上20未満の乱数です。したがって、Rnd() * 20 - 10は-10以上+10未満の乱数になります。一般的に、Rnd() * 2N - Nという表現は、ゼロ・プラスマイナスNの範囲の乱数値です。Turn()と組み合わせると、左右N度の範囲でランダムに方向を変える便利な技法になります。

それでは、hanabi(C)という名前を付けて保存してから、実行してみましょう。

新しく学んだ事項

- フローチャートの書き方
- イフ文(標準型、簡略型)
- イフ文の入れ子構造
- 左辺と右辺の関係を表す記号 ==, <, >=
- GetCountStep()
- Turn()
- Rnd()のいろいろな使い方

練習問題

5.1

次のようなルールのモデルを作ろう。

30ステップ目で左右に分かれたtoriたちが、40ステップ目でランダムな方向を向いて、以降その方向に飛び続ける。



ルールの書き方は一通りではありません。正解はいくつもあります。

5.2

Turn(D)がMy.Direction = My.Direction + Dと同じルールになることを確認しよう。



artisocでは、「=」は数学の等号ではありません。

大小関係を表す演算子、論理関係を表す演算子

第5章で、「==」や「<」「>=」という右辺と左辺の大小関係を表す記号の使い方を学びました。このような記号を演算子といいます。

上の3つ以外にも関係を表す演算子があります。ここでまとめておきましょう。通常の数学とはほんの少しだけ書き方が違うので注意が必要です。

$==$ 左辺と右辺が等しい

$<>, !=$ 左辺と右辺が等しくない(2通りの書き方があります)

$<$ 左辺が右辺より小さい(未満)

$<=$ 左辺が右辺より小さいか等しい(以下)

$>$ 左辺が右辺より大きい

$>=$ 左辺が右辺より大きいか等しい(以上)

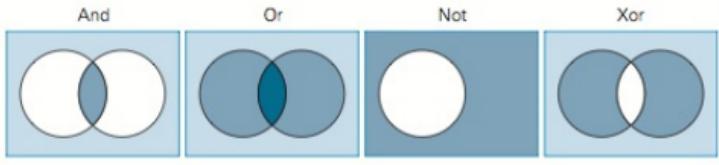
大小関係の演算子とともに、論理関係の演算子についてもまとめておきます。これらの使い方については第2部で学びます。

And かつ(論理積)

Or または(論理和)

Not ではない(否定)

Xor どちらかが真(排他的論理和)



これらの論理演算子や関係演算子を組み合わせることで、さまざまな条件式を書くことができます。

(光)

第6章

エージェントに周囲の環境を調べさせる

6.0 観察が肝腎です

- 近くに人がいると立ち止って集団ができるモデルを作りましょう
- いよいよ相互作用をモデル化します
- エージェント自身に自分の周囲にエージェントがいるか調べさせます
- エージェントに新しい変数を追加します
- artisocには便利な関数がたくさん用意されているのでモデル作りが簡単です

6.1 周囲の状況に応じた「自律的行動」

前章では、エージェントに本格的な「自律的な行動」をさせる前提として、「場合分け」の方法を学びました。前章の例では、ステップ数にしたがって、行動の場合分けを実行しました。しかしこれでは、エージェント自身が自分の置かれた周囲の状況を認識して行動を変えた、というわけにはいきません。「エージェントが自律的に行動している」とみなすためには、「エージェントの周囲の環境をエージェント自身が認識して、その認識如何にしたがって行動を選択する」というように、場合分けの条件を各エージェントに個別の条件にすることが重要です。

マルチエージェント・シミュレーションの特徴のひとつに、エージェントは全体についての知識は持たず、自分の周囲の状況だけを知ることができる、という設定方法があります。この方法を用いれば、エージェント毎に周囲の環境は異なりますから、周囲の環境に応じて異なる行動を各エージェントにとらせることができます。エージェントの自律性はいっそう強い意味になります。

6.2 自分の周りのエージェントたち

artisocに備わったエージェントの周囲の認識方法を利用して、周囲の環境によってエージェントの行動を変えさせる基礎を学びましょう。前章で学んだ「場合分け」の条件として、周囲の環境についての情報を用いるのです。

特にここでは、エージェントの周囲にいる他のエージェントを認識する方法の基本を学びます。[図6.1](#)のように、特定のエージェント（自分）の周囲にどのようなエージェントがいるかを認識するということは、(1)「周囲」の具体的な範囲（「視野の広さ」）、(2)周囲を見回して、そこにいる他のエージェントを「認識」したことを表す変数、をはっきりさせることができます。(1)については直感的に分かると思いますが、(2)については少し説明が必要かも知れません。周囲に、友達のPさんとTさん、それにPさんが飼っている猫のQ、Tさんが飼っている犬のVがいるとしましょう。周囲を認識するということは、自分の周囲に「Pさん、Q、Tさん、V」がいるとわかることです。周りにいる人間を認識するなら「P、T」、ペットなら「Q、V」です。したがって、周囲にいるエージェントを認識したことを表す変数のとる「値」は、数値ではなく、{P, T}, {Q, V}, {P, Q, T, V}といったエージェントの集合になります。このように、エージェントたち（エージェントの集合）を「値」とする変数を「エージェント集合型」変数と呼びます。

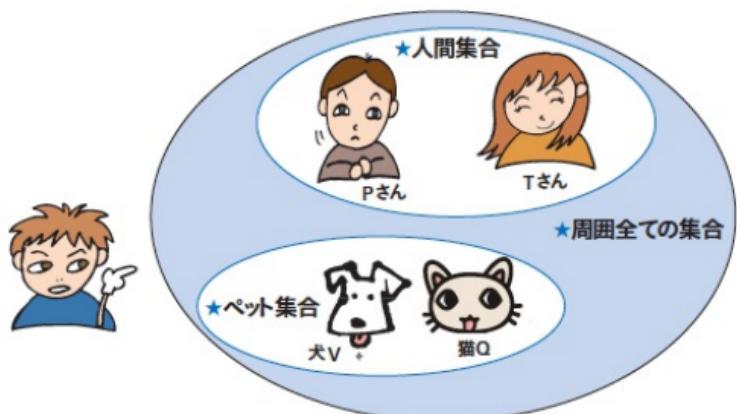


図6.1

6.3 新しい変数をエージェントに追加する

まず、今までの復習を兼ねて、全く新しいモデル作りの準備作業をしてください。空間の名前を hiroba(プロパティはデフォルト)、エージェント(種)はhitoとして、エージェント数を100に設定して下さい。

エージェントには、mawariという変数を追加して下さい。artisocが自動的に設定するエージェントの変数だけでは、周囲を認識させることはできません。mawariは自分の周りにどんなエージェントがいるかを調べた結果を記録しておくための変数です。なお、エージェントの変数名を、ここでは日本語mawariにしましたが、Directionのように英単語(たとえばneighborhood)でもかまいません。

変数の追加は新しい技法です。まず、hitoを選択して **hito** とハイライトしてから、**挿入** から **変数の追加** を選んで下さい。すると、変数を追加するための「変数プロパティ画面」が開きます。「変数名」のところにmawariと書き込み、「変数の型」をデフォルトの整数型ではなく、エージェント集合型に変えて下さい([図6.2左図を参照](#))。そして、**了解** を押して下さい。作業が正しく行なわれたならば、ツリーでhitoの下に(自動的に作られる変数に続いて)mawariが追加されているはずです([図6.2右図](#))。

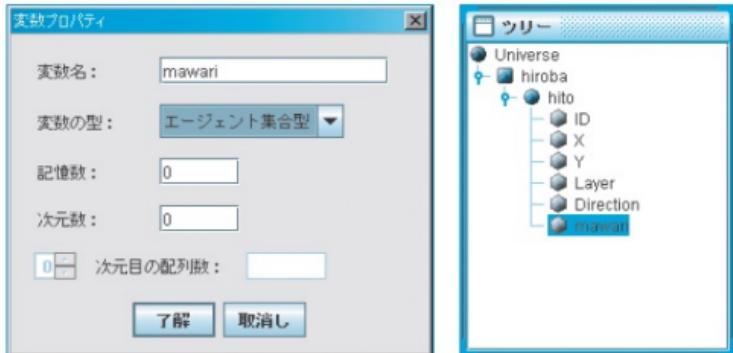


図6.2

出力設定(マップ出力)も、忘れずにして下さい。やり方を忘れた人は[3.4](#)を参照して下さい。このモデルは、tachibanaishiと名付けて保存して下さい。

6.4 周囲にいる人と立ち話をさせる

広場を歩いている人たちが、近くに集まると立ち止まる。

次のようにエージェントを行動させてみましょう。

1. hirobaにばらばら(ランダム)にいる
2. ランダムな方角を向く
3. 每ステップ、周囲(視野の広さ2)を観察する
4. 他のhitoが3人以上いるときには、そこで立ち止まる。いなければ「1」ずつ、歩き続ける

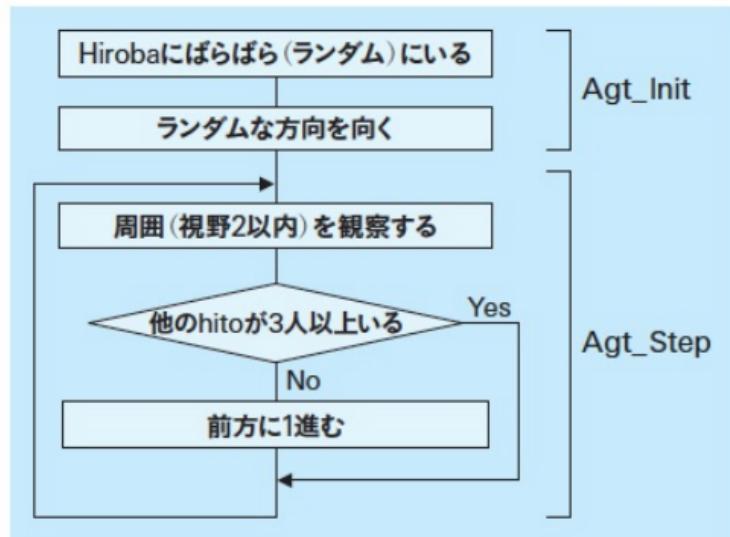


図6.3

1と2は、Agt_Init{}の中に書き込むルールです。3, 4がAgt_Step{}の中に書き込むルールで、周囲の環境に応じた行動の場合分けです。

ルールエディタには次のようにルールを書き込みます。新しい表現がいくつか登場しますが、すぐ後で、説明します。

```
Agt_Init{  
My.X = Rnd() * 50  
My.Y = Rnd() * 50  
My.Direction = Rnd() * 360  
}  
Agt_Step{  
MakeAllAgtsetAroundOwn(My.mawari, 2, False)  
If CountAgtset(My.mawari) >= 3 Then  
  
Else  
    Forward(1)  
End if  
}
```

では、上のルールを説明します。上半分(Agt_Init{ })はもう説明不要でしょう。hitoの初期の居場所や歩こうとする向きを、Rnd()という一様乱数発生関数を利用して、ランダムに設定しています。

後半のAgt_Step{ }部分を説明します。

1. まず

```
MakeAllAgtsetAroundOwn (My.mawari, 2, False)
```

が新しいルールです。これは、mawariというエージェント集合型変数に、視野2の範囲の周囲にいるエージェントたちを全て認識させる(覚させる)というルールです。括弧の中の最後のFalseは、自分自身を含めない認識方法を表しています。つまり、もし孤独状態に置かれていれば、Falseだと「誰もいない」と認識します。ちなみにこの部分をTrueと記述すると「自分しかいない」と認識します。

2. それに続く大きな構造であるIf XXXXX Then YYYYY Else ZZZZZ End ifは、前章で学んだ「場合分け」の基本構造です。条件XXXXXの部分にあるCountAgtset(My.mawari) >= 3が新しく登場した表現です。CountAgtset()は、括弧中のエージェント集合型変数に認識されている

エージェントの数を求めるための関数です。この関数のとる値は整数です。つまりこの場合、自分の周囲にいるエージェントが3以上ならYYYYYYというルールで行動し、そうでないなら(3未満なら)ZZZZZというルールで行動する、ということです。

3. ここでYYYYYYのルールが何も書かれていないことに注意して下さい。これは何もしない(じっとしている)ことに対応しています。このように、本来なら何かルールが書き込まれるはずの部分(ここでは
こう
YYYYYY)が空白なとき、「空文」である、と言うことがあります。

ルールを書き込んだら、上書き保存をし、実行してみて下さい。徐々にhitoエージェントが固まりを作つて、やがて全てのエージェントが立ち止まります([図6.4](#)参照)。

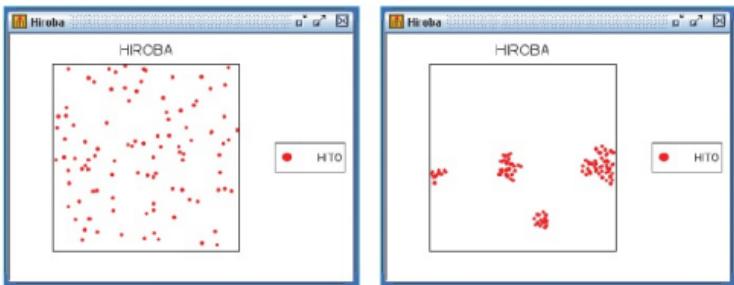


図6.4

6.5 パラメータを変えてみる

ここで、マルチエージェント・シミュレーションの醍醐味を少し味わってみましょう。

マルチエージェント・シミュレーションの目的の1つに、ミクロな状態(典型的にはエージェントの自律的行動の仕方)の変化がマクロな状態(典型的には空間での集団行動のあり方)の変化にどのような影響を与えるか、を調べることができます。ここで作ったモデルに従えば、たとえば「視野の広さや立ち話をするときの最少人数をオリジナルなモデルの設定(視野2、最少人数3)から変えることによって、立ち話集団の数

や規模がどのように変わるか」を調べることです。

このように、モデルの基本を変えないで結果がどのように変わるかを調べるために変化させる変数のことをパラメータと呼ぶことがあります。ここでは、視野の広さや立ち止まる最少人数がパラメータになっています。

tachibanaShiモデルのルールエディタを開いて、

1. 視野の広さを1に減らす、2のままにする、3に増やす
2. 最少人数を2人に減らす、3人のままにする、4人に増やす

という変更作業を全ての組み合わせについて行ない、その度に何回か「実行、終了」をしてみて、hirobaの全体的な様子の違いを観察しましょう。立ち話をする集団の数や大きさ、全員が立ち止まるまでの時間などに大きな違いが現れてくることが容易に見て取れだと思います。

ところで、注意深く観察しないとわかりにくいのですが、このシミュレーションでは、いったん立ち止まつてもまた歩き出すhitoがときどきいます。これは、各エージェントが毎ステップ自分の周囲を観察して、行動を変えるルールになっているために、多数のエージェントのルールが次々と実行されていく過程で、あるステップでは条件が満たされていたのに、次のステップでは条件が満たされなくなる状況が生じ得るからです。

6.6 モデルを複雑にしよう

自律的行動を「周囲の認識」と「場合分け」とから表す基本のまとめです。tachibanaShiモデルに次のような変更を加えましょう。今まで学んだ技法を総動員します。

広場には人だけでなくペットも走り回っている。

1. 新しくpetというエージェントを100匹追加します
「ツリー」の中の空間「hiroba」の下にhitoを作ったのと同じように作業します。



挿入 メニューから エージェントの追加 を選択します。

2. 出力マップにpetエージェントの出力を追加します。

新しい出力マップではなく、hitoを出力させたマップに追加する必要があります。

この作業は初めてですから、すこし丁寧に説明します。まず 設定 メニューから 出力設定 を選択すると出力項目リストが現れるので(hirobaがハイライトされているはずです) 編集 を選択します(追加 ではありません)。「マップ要素リスト」にpetを追加します。追加 を選択すると、「要素設定」ウィンドウが開きます。出力対象はhitoではなくpetを選択して下さい。

3. petエージェントは初めhirobaにランダムにいます。

4. petの走る方向は毎ステップ「ランダム」です。

5. petは毎ステップ「1」だけ走ります。

このように修正したモデルをtachibana(B)という新しい名前で保存して下さい。実行しましょう([図6.5](#))。

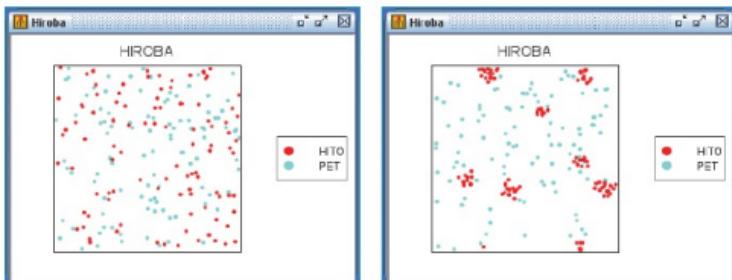


図6.5

[6.7 1種類のエージェントだけに注目する](#)

実は、tachibana(B)モデルでは、hitoが認識している周囲のエージェントはhitoだけでなくpetもいます。そのため、hitoとpetを合わせた数が一定以上だと、そこでhitoは止まってしまいます。そこで、petを無視して、他のhitoがそばにいるかどうかだけを気にするようにhitoエージェントの行動ルールを変えましょう。そのようなルールが、MakeOneAgtsetAroundOwn()です。上で学んだMakeAllAgtsetAroundOwn()に似ていますが、この新しい認識方法は1種類のみのエージェント種を認識します。ここではhitoだけを認識させることにしましょう。

```
MakeOneAgtsetAroundOwn(My.mawari, 2, Universe.hiroba.hito,  
False)
```

のように表現します。改行せずに、続けて入力して下さい。1行におさまらない場合は、artisocが自動的に折りかえしてくれます。なお、「.」の前後に空白(スペース)を入れてはいけません。

ここで、hitoエージェント種をUniverse.hiroba.hitoと表している点に注意して下さい。「Universeの下に作ったhirobaという空間で行動するhitoエージェント種」と厳密に表記する必要があります。

tachibana(B)モデルの周囲のエージェントを認識する部分を新しいものに変えて下さい。それをtachibana(C)という名前で保存して、実行して下さい(なお、視野と最少人数はもとの2と3に戻して下さい)。petは無視されていますから、hitoの行動はもともとのtachibanaモデルと同じになっているはずです。確認しましょう。ちなみに、私個人としては、ペットを無視して人間とだけ立ち話をする人よりは、人間だけでなくペットも(ペットだけでも)自分のそばにいると立ち止まるような人の方が好きですが。

なお、MakeAllAgtsetAroundOwn()やMakeOneAgtsetAroundOwn(), CountAgtset()のように、特定の行動をさせるために予め用意されているルールのことを、artisocの「組み込み関数」といいます。組み込み関数にはさまざまな種類のものがあります。既に用いている

Forward(), Turn(), Rnd(), GetCountStep()なども組み込み関数です。

ところで、これからいろいろな組み込み関数の使い方を学んでいきますが、既に上で使ったMakeAllAgtsetAroundOwn()やMakeOneAgtsetAroundOwn()のように長くて似た名前の関数群があります。そこで、入力の手間を省き、ミスをなくすために、artisocではMy.の「.」を入力したときのような入力

支援の機能が備わっています。たとえばルールエディタの中でMakeと入力してからコントロールキーとスペースキーを同時に押すと、Makeで始まる組み込み関数の一覧が現れます。My.変数の選択と同じ要領で選択することができます。

6.8 「もっと現実的に」はどれだけ本質的か

tachibanaShiモデルでもtachibanaShi(B), tachibanaShi(C)モデルでも、hitoは最初に歩く(走る)方向が決まるとずっとそのまま真っ直ぐ進みます。「等速直線運動」ですから、これでは不自然ですね。他方、petは毎ステップ全方位にランダムで向きを変えますが、これも不自然です。そこで、tachibanaShi(C)モデルのルールを少し変えてみましょう。

hitoは毎ステップ左右15度の範囲でランダムに、petは毎ステップ左右30度の範囲でランダムに進行方向を変える。hitoは毎ステップ「1」移動するが、petは毎ステップ「2」移動する。

hitoとpetのルール変更は独力で試みて下さい。



Turn(Rnd() * 20 - 10)は左右何度?

この新しいモデルをtachibanaShi(D)というファイル名で保存して下さい。

では実行してみましょう。petやhitoの動き方に変化がでたことはすぐに気がつくと思います。hitoはぶらぶら歩きに、petは活発に走り回る感じになったでしょう。たしかに、このように修正すると、不自然さが減りました。それでは、hirobaの全体的な特徴には大きな変化が生まれましたか。そんなに大きな違いはなかったはずです。

このモデルで本質的な部分は、エージェントの細かな動き方ではなく、周囲の環境をどのように認識し

て行動に結びつけるか(視野の広さ, hitoだけか hitoもpetもか, どれだけそばにいれば立ち止まるか)にあることがわかったと思います。「シミュレーションはなるべく現実に近づけた方がよい」という考え方があります。しかし、どんな場合でも現実らしいモデルを作らないといけない、というわけではないことは、この例でわかつてもらえたでしょう。

新しく学んだ事項

- エージェントに変数を追加する方法
- エージェント集合型変数
- MakeAllAgtsetAroundOwn()
- MakeOneAgtsetAroundOwn()
- CountAgtset()
- 組み込み関数
- 組み込み関数の入力支援機能
- 出力設定の編集

練習問題

ここではモデル作りではなく、ルール表現についての頭の体操をしてもらいましょう。

6.1

自分を含めるか含めないか。

```
MakeAllAgtsetAroundOwn(My.mawari, 2, False)  
If CountAgtset(My.mawari) >= 3 Then
```

上の記述と下の記述が同じルールになるためには、下の「?」のところに何を書き込めばよいだろうか。

```
MakeAllAgtsetAroundOwn(My.mawari, 2, True)  
If CountAgtset(My.mawari) >= ? Then
```

6.2

何もしない場合を明示するかしないか。

```
If CountAgtset(My.mawari) >= 3 Then  
Else  
    Forward(1)  
End if
```

上の記述と下の記述が同じルールになることを確認しよう。

```
If CountAgtset(My.mawari) < 3 Then  
    Forward(1)  
End if
```

練習問題6.2では、同じルールでも下の記述の方が短くなっていますが、必ずしも短ければよいというわけではありません。わかりやすい表現が最も大事な点です。上の記述では、周囲のエージェントが3以上の場合と3未満の場合とでルールがどのように異なるのかがはっきりとわかります。何もしない場合を「空文」としてわざわざ明記してあるからです。

酔っぱらいは自律的な主体なのか

確率過程のモデルに昔から有名な「酔歩(random walk)」というのがあります(私なら千鳥足と訳しますが)。つまり、酔っぱらいが現在立っている地点から次の時点にどこに向かうのか、本人は真っ直ぐ歩いているつもりでも、酔っぱらっているので無意識のうちに確率的に決まるような状態をモデル化したものです。このように、将来が決定論的には決まらない状況を、確率を用いて表現することがよくあります。

マルチエージェント・シミュレーションの技法でも、エージェントの行動を決めるルールの中で確率を頻繁に使います。特に、エージェントに自律性を持たせようとするときに、条件分岐とともに、確率(乱数)を利用します。「ランダム」と自律性とが深く結びついているのです。

それでは、千鳥足の酔っぱらいは自律的主体なのでしょうか。次の時点にどこに向かうのかを合理的に決めていないのは確かです。しかし、仮に「しらふ」で真っ直ぐに歩いているとしても、次の一步をどこにするのか合理的に判断している人はいないでしょう。無意識に真っ直ぐ歩いているはずです。意識しないと次の一步を踏み出せない人は、病気かリハビリ中に違いありません。結論的には、酔っぱらいも自律的主体であると考えることにしています。

奴隸のように(労働者のように?)主人(雇用主?)の命令に従って素直に働く人間は、因果論的に記述できるので、自律的主体ではありません。サイコロを振って意思決定をする(かのように行動する)人間は、他人が(神でさえ?)左右できないので、自律的主体と見なされるのです。

ちなみに、直線的に飛ぶガに比べて、チョウはヒラヒラ飛びますが、トリに見つかりやすい昼間に行動するチョウは、ヒラヒラ飛ぶ(自分自身でも次の時点どこに向かうのかわからない)ために、かえってトリに食われにくいそうです。夜に活動するガはヒラヒラ飛ぶ必要がなかったというわけです。チョウとガの行動パターンの違いを進化ゲームの問題としてマルチエージェント・シミュレーションのモデルを作ってみるのも一興です。もっとも人間世界では、夜の方が、千鳥足やチョウをよく見かけますが。

第7章

モデルの設定値をモデルの外部から操作する

7.0 「神様」になりましょう

- モデルを外部から操作することができます
- 操作するためのコントロールパネルを作ります
- モデルの内部を外部からの操作に対応できるように変えます
- Universeが重要な「神様」の役割をします
- 「神様」にエージェントを「創造」させ、初期配置させましょう
- 「繰り返し」をさせる「フォ文」を学びます
- ツリーにない一時的変数を使います

7.1 モデルの条件を簡単に設定し直したい

前章まで、

1. エージェントに行動させる(具体例としては「動く」「ルール」「方向」と「速さ」を書き込む)
2. エージェントに自律的行動をさせる(具体的には「場合分け」で異なる行動様式を選択させる)
3. エージェントに周囲の環境を調べさせる(具体例としては周囲のエージェントを認識させる)
4. 周囲の環境の状態に応じて、行動を変える(以上の総合)

というマルチエージェント・シミュレーションの基本的考え方を学びました。

その際、エージェントのプロパティ・ウィンドウを開いて、エージェント数を変えたり、エージェント・ルールエディタを開いて、さまざまな条件の数値を変えたりして、異なる状態でシミュレーションを実行し、違いを比較しました。言い換えれば、マルチエージェント・シミュレーションを本格的に行なうことは、基本的なモデル

の中の一部をいろいろ変えてみながらシミュレーションを実行して結果に現れる変化を観察することなのです。

artisocは、さまざまな設定を簡単に変えられる機能を備えています。この章では、いちいちエージェントのプロパティ・ウインドウやルールエディタを開くことなしに、さまざまな設定を変えられるようにする方法の基本を学びます。具体的には、「コントロールパネル」という操作盤を作ることによって

1. エージェント数をモデルの外から(コントロールパネルで)設定する
2. 「場合分け」の条件をモデルの外から(コントロールパネルで)設定する

という外からの操作が可能なモデルを作る方法を学びます。これは、エージェントが相互作用する空間より上位にあるUniverseのレベルでさまざまな操作をする技法です。相互作用をするエージェントたちの世界を見渡す高見に立って、いわば「神様」の立場から、いろいろと介入できるのです。

7.2 エージェント数を自由に変える

まず、エージェントのプロパティとして設定してきたエージェントの数をモデルの外側から操作する方法を学びます。

その前にとりあえず、前章で作ったtachibanaishiモデルを、また最初から作る準備作業をして下さい。手っ取り早く慣れるには繰り返しが大切です。

その際、エージェントとしては、hitoを作りますが、エージェント数は0のまま(デフォルト値)にしておいて下さい。エージェント数は、モデルの外部から指定するからです。

次に、hitoのルールエディタには、[6.4](#)と同じルールを書き込んで下さい。この新しいモデルを、tachibanaishi2と名付けて保存して下さい。

artisocでモデルの外から設定する数だけエージェントを生まれさせるためには、

1. Universeにエージェント数を表す変数を作る
2. その変数をモデルの外から操作できるように、コントロールパネルを作る
3. Universeのルールエディタに、コントロールパネルで指定したエージェント数だけのエージェントを生成させるルールを書き込む

という3手順が必要です。

それでは、エージェント数を表す変数(整数型)をninzuとして、1と2の作業をしましょう。

1. まず、Universeをクリックして **Universe** とハイライトし、**挿入** メニューから **変数の追加** を選んでください。変数を設定するための「変数プロパティ」ウィンドウが開きます。変数名の欄にninzuと書き込み、変数の型が「整数型」になっているのを確認したら、そのまま **了解** を押して下さい。ツリーのUniverseに直接つながる変数ninzuができましたね。
2. つぎに、いま作ったninzuの値をモデルの外から設定できるようにしましょう。**設定** メニューから **コントロールパネル設定** を選んで下さい。「ユーザ設定項目リスト」が出てきます。ここで **追加** を押すと、新しい「ユーザ設定項目」の画面が出てきます。「コントロール名」に、NINZUと書き込みましょう。そして、そのすぐ下の「設定対象」で、ninzuを選びます。すると、インターフェースが選べるようになります。いま設定したいのは人数ですから、「スライドバー」か「直接入力」が適しています。今回は「スライドバー」を使いましょう。「範囲」は変数を設定できる幅を示しています。10から200を入れておきましょう。「目盛り間隔」は1(デフォルト値)のままでも構わないでしょう(図7.1参照)。**了解** を押し、ウィンドウを閉じ、続いて **閉じる** を押して「ユーザ設定項目」も閉じて下さい。



図7.1

これで、コントロールパネルの設定が終わりました。一度上書き保存してから、実行ボタンを押してみましょう。エージェントを生成するルールを書いていないので、hirobaの中にはエージェントが存在していません。しかし、コントロールパネルが出てきたと思います(図7.2参照)。スライドバーのつまみをクリックしたまま横に動かしてみてください。数字が10から200まで動いたでしょうか？ちなみに、スライドバーはキーボードの「←」「→」の矢印ボタンでも操作できます。試してみて下さい。このスライドバーの値にしたがって、ツリーの中のninzuという変数の値も変わっていくのです。ここまでのお作業で、モデルの外部からモデルの中の変数を操作するインターフェースが完成しました。停止ボタンを押して下さい。



図7.2

7.3 モデルの中でエージェントを生まれさせる

それでは前節で示した最後の作業3に取りかかりましょう。つまり、ninzuという変数の値の数だけエージェントを生成するルールを書きます。まず、どのようなルールにするかを明記してから、新しい文法などを説明します。

1. Universeのルールエディタを開きます。
2. Univ_Init{ } の間に必要なルールを書き込みます(それ以下の部分はとりあえず無視して下さい)。Univ_Init{ }に書き込まれたルールは、シミュレーション実行時の最初に1回だけ実行されるルールです。
3. ルールのポイントはninzuぶんのエージェントをどうやって生成するかです。それが以下の技法です。

```
Univ_Init{
Dim i As Integer
```

```
For i = 0 To Universe.ninzu - 1
    CreateAgt(Universe.hiroba.hito)
Next i
}
```

わずか4行のとても短いルール表現ですが、ここには新しいものがたくさんあります。

1. まず、Dim i As Integerという表現に注目して下さい。これはiという変数を整数型として(As Integer)作って下さいというルールです。ルールエディタのセクション(この場合Univ_Init{ })で「そこだけで一時的に使う変数」が必要な場合、セクションの冒頭であらかじめ宣言しておきます。わざわざツリーの中で作る必要のない「使い捨て変数」です。
2. 次に、

```
For i = 0 To Universe.ninzu - 1
    XXXXX
Next i
```

という構文に注目して下さい。これは、XXXXXという作業を0からUniverse.ninzu - 1回(つまり合計Universe.ninzu回)繰り返すということです(ここでは、For i = 1 To Universe.ninzuと書いても正しく実行されるのですが、artisocではさまざまな設定が1からではなく0から始まっているので、他の箇所での汎用性を考慮して、なるべく0から始まるように繰り返し文を書く習慣を身につけて下さい)。なお、Universe.ninzuというのは、「Universeの下にあるninzuという変数」を意味しています。曖昧さをなくすための表記法です。この繰り返しをさせるルール表記を「フォ文」と呼びます(理由は明白ですね)。

3. 繰り返す作業XXXXXはCreateAgt(Universe.hiroba.hito)ですが、これはUniverseの下のhirobaという空間上で定義されているhitoというエージェント種を1体だけ生み出す、というものです。それをninzu回繰り返すわけですから、ninzu人のhitoが生成されることになります。

それではいよいよ実行することにしましょう。コントロールパネルは残っているはずです。hirobaに登場するhitoの数をコントロールパネルで適当に設定しましょう。上書き保存してから実行ボタンを押します。Univ_Init{ }で生成されたninzu人のhitoは、Agt_Init{ }のルールで初期配置されます。

7.4 モデルの中のパラメータを簡単に変える

このモデルでは、hitoエージェントの行動は、自分の視野の広さと視野に入る他のhitoの数で決まります。このモデルでは、視野の広さや立ち止まるときの最小人数は全てのhitoエージェントに共通です。そこで、視野の広さをshiy a、行動を左右する人数をnakamaという変数で表すことにして、上のninzuと同様に、モデルの外からコントロールパネルで設定できるようにしましょう。ここで、注意を要するのは、必ずUniverseの直下にこれらの変数を追加する必要がある、という点です。

作業の手順を簡単に説明します。

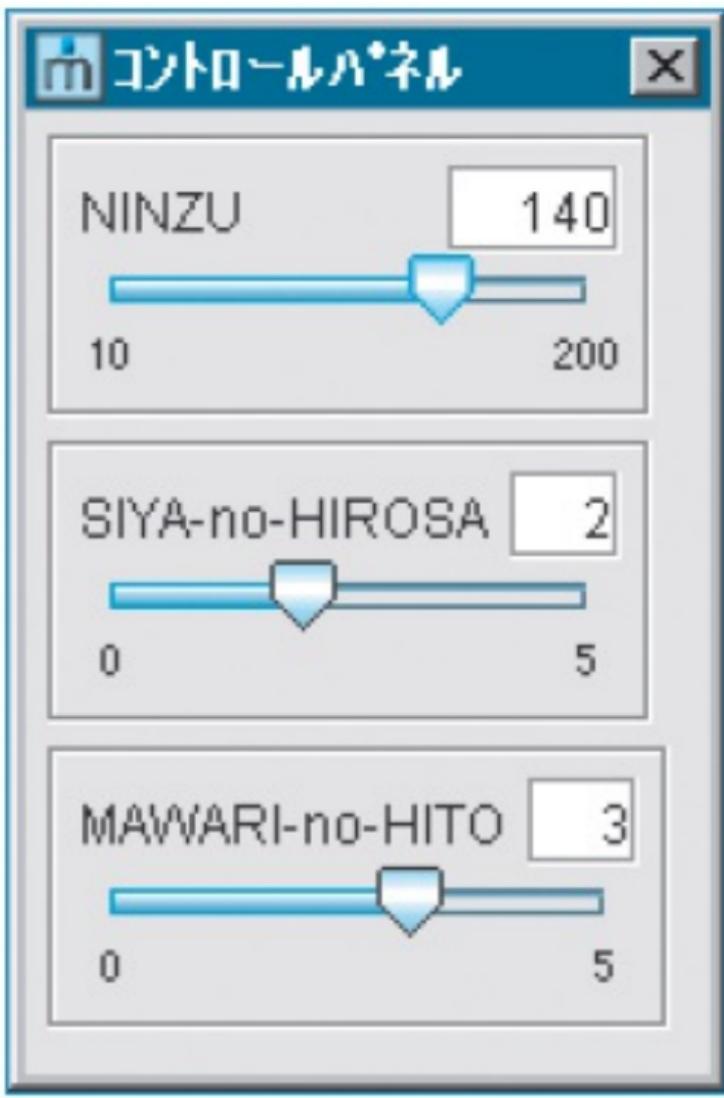
1. Universeに直接「shiy a」と「nakama」という名前の整数型変数を追加します。追加方法は7.2のninzuの追加と同じです。
2. 設定 から コントロールパネル設定 を開きます。「ユーザ設定項目リスト」には、先ほど作った「NINZU」という項目があります。下の 追加 を押して、新たな項目「SIYA-no-HIROSA」(shiy aを設定対象に)と「MAWARI-no-HITO」(nakamaを設定対象に)を設定しましょう。どちらもスライドバーで、0から5の範囲を選べるようにコントロールパネルを作つて下さい。やり方は7.2と同様です。目盛り間隔はデフォルト値(1)のままにしておいて下さい。
3. 次に、hitoエージェントのルールエディタを開いて下さい。そして、視野の広さや条件文の中の整数値を、Universe.shiy aとUniverse.nakamaという変数に置き換えて下さい。具体的には、次の2カ所です。

```
MakeAllAgtsetAroundOwn (My.mawari, Universe.shiya, False)
```

(MakeOneAgtsetAroundOwn()を用いている人も、同じように書き直して下さい)

```
If CountAgtset(My.mawari) >= Universe.nakama Then
```

それでは、tachibanashi2(B)という名前を付けて保存してから、実行しましょう。なお、コントロールパネルが現れても、表示が不完全な場合には、右下のつまみをマウスで動かして、パネルが全部現れるように調整して下さい。



シミュレーションの最中に、コントロールパネルでshiy aやnakamaの値を変化させてみましょう。hitoの行動パターンが大きく変化する(急に立ち止まつたり、また歩き出したりする)ことが観察できるはずです。モデルの中で(具体的にはエージェント(種)のルールエディタの中で), Universe.shiy aやUniverse.nakamaの値が毎ステップ参照されるルールになっているために、実行途中で変化させると、すぐにそれが行動パターンの変化につながるのです。

7.5 エージェントを生まれさせて、初期配置する

今まで、エージェントの初期配置などの初期値設定は、エージェント自身のルールで行なってきました。いわば、エージェントが個別に自分の居場所を最初に決める、という方式です。これに対して、(神様が)エージェントを生まれさせるのと同様に、(神様が)生まれたエージェントを配置するという方式も可能です。やはり、Universeのルールとして設定します。この方法をこれから学びます。

ここで取り上げるのは、エージェントをシミュレーションの最初にランダムに配置する作業です。結果は同じになりますが、モデル作りの発想は大きく異なります。

tachibanaShi2(B)モデルを用いて説明しましょう。

1. hitoエージェントのルールエディタを開いて、Agt_Init{ }の中にある

```
My.X = Rnd() * 50  
My.Y = Rnd() * 50
```

の2行を削除します。My.Direction = Rnd() * 360は、そのまま残します。

2. Universeのルールエディタの、Univ_Init{ }のかっこ内の部分に次のルールを付け加えます。

```
Dim hitobito As Agtset  
MakeAgtset(hitobito, Universe.hiroba.hito)
```

```
RandomPutAgtset(hitobito)
```

1行目は、hitobitoという変数をエージェント集合型変数として(As Agtset)、これから一時的に使うことを宣言しています。2行目は、hitoエージェント種をhitobitoという変数のなかにしまう操作です。これにより、既に書き込まれているルールに従って生成されている全てのhitoエージェントがhitobitoに含まれます。そして3行目は、hitobitoというエージェント集合型の変数にしまわれている全エージェントをランダムに配置します。

ここで、エージェントを集める関数について補足しておきます。この例では、モデルには、エージェントは1種類(hito)しかありません。このような場合、あるいはエージェント種はいくつかあるが、全てのエージェント種について存在しているエージェントを全て集める場合、MakeAgtsetSpace()という関数が使えます。上の場合だと

```
MakeAgtsetSpace(hitobito, Universe.hiroba)
```

というふうに使います。エージェント種を指定しないで、空間名だけを用いている点に注意して下さい。

なお、Dim hitobito As AgtsetとかDim i As Integerのような一時的に使う変数の宣言文は、ルールの途中にあるとエラーの原因になる場合があるので、セクション(この場合Univ_Init{ })の冒頭にまとめるようにして下さい。

結局、Univ_Init{ }に書き込むルールは全体として次のようになります。

```
Univ_Init{
Dim i As Integer
Dim hitobito As Agtset

For i = 0 To Universe.ninzu - 1
  CreateAgt(Universe.hiroba.hito)
Next i

MakeAgtsetSpace(hitobito, Universe.hiroba)
RandomPutAgtset(hitobito)
}
```

上のルールでは、MakeAgtset()ではなくMakeAgtsetSpace()を使っています。

さて、エージェントのルールとしてできることをなぜわざわざUniverseのルールとして書き換えてみたのでしょうか？書き換えるメリットは何でしょうか？上で削除したやり方では、空間の大きさが50×50に限って、正しいルールです。これに対して、新しく登場したRandomPutAgtset()は空間の大きさをあらかじめ指定していません。したがって、ルールを全く変えずに、空間の大きさを自由に変えることが可能になるのです。この例では結果は同じですが、やがて、複雑なモデルを作るときに、どのような場合にどちらの方法が望ましいか分かってくるでしょう。

書き換えたモデルをtachibanaishi2(C)という名前で保存して、実際に以前と同じように動くかどうか、実行して確かめて下さい。

なお、tachibanaishi2(C)モデルは、次章でそのまま使います。下の練習問題でモデルを修正しますが、保存する場合は別の名前(たとえば、tachibanaishi2(renshu))を付けて保存して下さい。

新しく学んだ事項

- フォ文
- 一時的変数の定義(型宣言文)
- CreateAgt()
- MakeAgtset()
- MakeAgtsetSpace()
- RandomPutAgtset()
- Universe変数の追加
- コントロールパネルの設定
- Univ_Init{ }へのルールの書き込み
- コントロールパネルを操作してモデルのパラメータを変える
- シミュレーション実行中にコントロールパネルを操作する

練習問題

復習です。次のようにモデルを修正して、Universeの使い方やコントロールパネルの設定に慣れて下さい。

7.1

生成したいhitoエージェント数を自由に設定できるように設定を変える。



コントロールパネルを「直接入力」に設定できます。ユーザ設定項目リストの編集機能を使います。

7.2

前章で作ったtachibana(B)モデルを参考にして、tachibana(B)モデルにpetエージェントを追加して、ペットの数やペットの移動速度をモデルの外部から設定できるように修正する。petを適当に動き回らせる。



Universeでpetを生成するだけでは不十分です。petのルールを適当に書き込んで下さい。出力設定も必要です。

ツリーの中の変数とルールの中の変数

変数には、大きく分けて2つの種類があります。ひとつは、ツリーに表示される「正式な」変数です。それぞれのエージェントが、それぞれ異なる値を持ち歩かなければならないようなタイプの変数にはこちらの変数を定義してやります。第6章で追加したmawariは、これに該当します。もうひとつはルールの中で定義される「一時的な」変数です。これは、ルールを記述する上で必要に応じて作成される変数で、ツリーには表示させずに用います。ルールの中で「Dim(変数名) As(変数の型)」の形で定義します。変数をルールの中で使ったあと、その値を保存する必要のないようなタイプの変数にはこちらの変数を定義してやります。第7章で導入したhitobitoがこれに該当します。

両者の最大の違いは、「正式な変数」はそれぞれのエージェントが、その変数のそれぞれの値を恒常的に備える(Myで識別します)のに対し、「一時的な変数」はそのとき処理しているエージェントについてのみ、その変数の値が設定され、処理の順番が次のエージェントにうつった時点で、その値は初期化されてしまうという点にあります。それぞれに特徴があるので、うまく使い分ける必要があります。値の保存の必要があれば、ツリーの中に変数を設定し、不要であればルールの中で定義してやって下さい。

また、「正式な」変数においても、「一時的な」変数においても型は、必ず必要です。型の指定は、前者はプロパティ・ウインドウで、後者はルールエディタの中で行ないます。

(保)

第8章

シミュレーションの過程をいろいろ出力したり、管理したりする

8.0 見ているだけが能じゃない

- シミュレーションの出力はマップだけではありません
- 時系列グラフに過程を出力させる方法を学びます
- 一つの時系列グラフにいろいろな指標を出力できます
- シミュレーション実行を管理することもできます
- 終了条件を設定しましょう
- 終了したらコンソール画面で知らせましょう

8.1 シミュレーションの経過をもっと知るには

シミュレーションの実行過程は、今まで、空間をマップ出力して、観察してきました。しかし、この方法だけでは、状態が刻一刻変化する状態は実感できても、どのように変化したかを正確に記録することはできません。

この章では、シミュレーション実行中のモデルの状態を、マップ出力以外の方法で出力する方法を学びます。具体的には、モデルの全体的特徴を把握するために集計作業を行ない、その結果を時系列グラフとして見ることを可能にする方法を学びます。

また、シミュレーションを管理する方法の初步も学びます。具体的には、ある条件が満たされたらシミュレーションを終了させる方法を学びます。

8.2 モデルの中で集計させる

ここでは、前章で作ったtachibanaishi2(C)モデルを使って、モデルの状態をどのように把握するのか、それをどのように表すのか(出力するのか)を学びましょう。まず、tachibanaishi2(C)モデルのファイルを開いてから、それをtachibanaishi3という名前を付けて保存して下さい。この章では新しい名前のモデルを使います。

シミュレーションの各ステップで、どれだけの人数が立ち止まっているのかを調べる。

以下の4手順で完了です。

1. Universe の下にtachidomariという状態を調べるための変数(整数型)を追加します。
おなじみになったはずの「追加」の操作をして下さい。
2. tachidomariを「時系列グラフ」として出力するように設定します。

設定 メニューから 出力設定 を押して「出力項目リスト」を開きます。そしてデフォルトの マップ出力 ではなく、 時系列グラフ を選択します。それから 追加 を押すと「時系列グラフ設定」の画面が開きます。グラフ名もグラフタイトルもTACHIDOMARIと書き込んで下さい(図8.1左図参照)。

「時系列グラフ要素リスト」の小窓の中の 追加 を押して下さい。「要素設定」画面が開きます。ここで時系列グラフで何を表すのかを指定します。要素名にはTachidomariと書き込み、出力値の項目には直接、出力したい変数であるUniverse.tachidomariをそのまま書き込んで下さい。線の太さや線の色は好みに変えてかまいません(図8.1右図参照)。作業が全て終わったら 了解 を順次押して、「出力項目リスト」(TACHIDOMARIが追加されているはずです)を閉じて下さい。



図8.1

3. Universeのルールエディタを開いて、Univ_Step_Begin{ }の中に次のような変数の初期化を毎ステップの冒頭に行なうようにルールを書きます。

```
Univ_Step_Begin{
    Universe.tachidomari = 0
}
```

Univ_Step_Begin{ }の中には、毎ステップの冒頭に実行するルールを書き込みます。こうすることによって、tachidomariという立ち話しているエージェントの数を各ステップでゼロに初期化して、そのステップで何人のエージェントが立ち話しているのかを数えることが可能になります。

4. hitoのルールエディタを開いて、次のようにして、自分自身が立ち止まつていれば、hiroba全体で立ち止まっているhitoの人数を1だけ増やすルール(Universe.tachidomari = Universe.tachidomari + 1)を挿入します。なお、薄字の部分は既に書き込まれているはずのルールです。

```
If CountAgtset(My.mawari) >= Universe.nakama Then
    Universe.tachidomari = Universe.tachidomari + 1
Else
```

Universe.tachidomariの値は、毎ステップの冒頭、Univ_Step_Begin{ }に書き込んだルールによって、ゼロに初期化されますから、各エージェントについてステップ毎に、自分自身が立ち止まっているかどうかを判断し、立ち止まっているhitoエージェントの数を集計していることになるのです。

ここで注意が必要です。hiroba全体で立ち話をしている人を数えるのは、個々のhitoエージェントではなく「神様」であるはずです。したがって、本当ならUniverseのルールで行なうべき作業です。もちろん、そのようにルールを書くことは可能ですが、ここでは、hitoが「神様」に対して「自己申告」している、と考えて下さい。これにより、同じような作業(つまり止まっているかどうかの確認)を個々のエージェントのレベルとUniverseのレベルの両方で毎ステップ同じようなルールを実行する無駄を省いています。

ただし厳密に言うと、この「自己申告」方式と、「神様の集計」とでは値が変わることがあります。前者だと、各エージェントが自身のルールが実行されているときの判断によって集計するのに対し、後者ではステップの最終時点での状態を大域的に見渡しているからです。Univ_Step_End{ }のセクションで簡単に集計するためにはまだ学んでいない技法を使います。これは第13章で学びます。

以上の変更を終えたら、上書き保存しましょう。では実行ボタンを押して、エージェントが動き回るマップと時系列グラフの両方が出力される様子を確認して下さい([図8.2](#))。

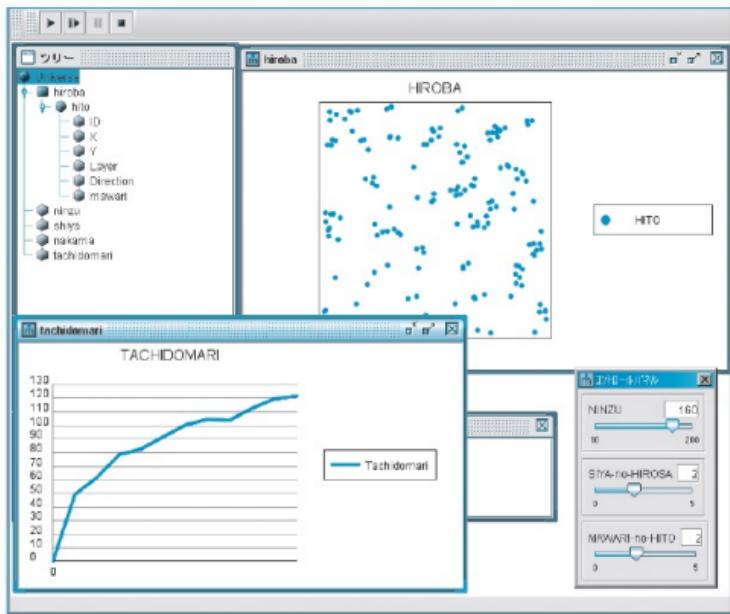


図8.2

8.3 時系列グラフの賢い利用法

時系列グラフを出力させる方法をうまく利用すると、ひとつの時系列グラフにいろいろな数値を表示することも可能です。時系列グラフの出力設定を開いて、次のような作業をしましょう。

立ち止まっているエージェント数の割合(パーセント)を出力する。

時系列グラフを編集することにより、新しい要素を追加します。割合をwaraiという名称でグラフ化します。要素設定画面の上で、

```
100 * Universe.tachidomari / Universe.ninzu
```

という計算結果を直接出力値としてグラフ化できます。Universeのルールエディタの中で、わざわざ出力したい値を計算させる必要がありません(図8.3参照)。ここでtachibana3(B)という名前を付けて保存して、実行してみましょう。



図8.3

同じ時系列グラフに出力要素を適当な方法で追加すれば、いろいろな値を見ることができます。

もっとも、ここで少し注意が必要です。たとえば、0と100の間を変化するグラフ(たとえばパーセント)と、0と10000の間を変化するグラフ(たとえば人数)をひとつのグラフで表示すると、後者の目盛りが縦軸になりますから、パーセントのグラフの変化がとても小さくなり、見づらくなります。このような場合、複数の表示数値があり大きく違わないように、補正する必要があります。上の例では、後者を100単位で表す(生の数値を100で割る)とよいでしょう。

8.4 自動的にシミュレーションを終了させる

今まで作ってきたモデルは、停止ボタンを押さない限り、シミュレーションはいつまでも実行され続けます。しかし、tachibanashiモデル（とその修正版）では、やがて、全てのエージェントが止まってしまい、その後、いつまでシミュレーションを続けても変化ありません。そこで、モデルの中で、シミュレーションを終了させる方法を学びましょう。

シミュレーションを終了させる条件を、全てのエージェントが止まったとき、にしましょう。この条件を毎ステップの最後にチェックすることにします。その場合、Univ_Step_End{}の中にルールを書き込みます。

```
Univ_Step_End{
If Universe.tachidomari == Universe.ninzu Then
  ExitSimulation()
End if
}
```

ExitSimulation()というのは、シミュレーションを終了させるルールです。hitoの数(ninzu)とtachidomariの数とが等しい(==)ときに限り、このルールが活性化します。ここでtachibanashi3(C)という名前で保存してから、実行してみましょう。自動的に終了したでしょうか。

8.5 終了時にartisocに一仕事させる

8.4で、モデルを一定条件が満たされれば、自動的に終了させる方法を学びました。シミュレーションの終了は、操作ボタンの様子が変わることでわかりますが、あまりはっきりしません（[図8.4](#)参照）。

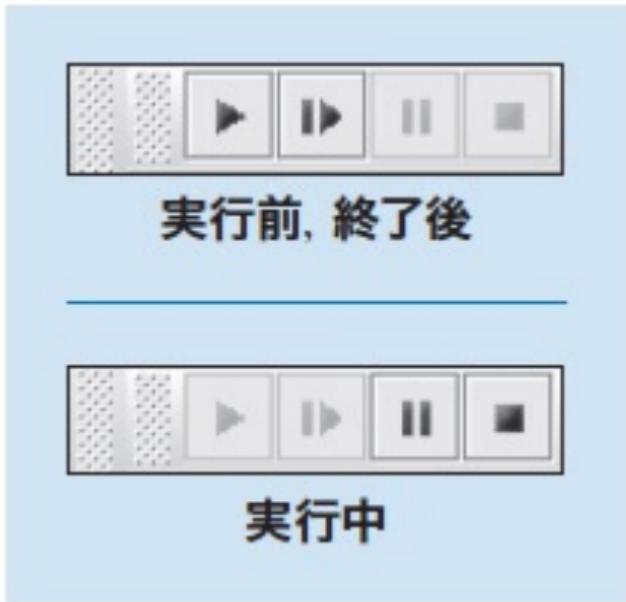


図8.4

そこで、終了時に、何ステップで終了したかをコンソール画面に書き出してみましょう。その方法を以下で学びます。

ひとつ的方法は、シミュレーションの終了時に実行されるルールであるUniv_Finish{}を利用してしまします。

```
Univ_Finish{
PrintLn("Simulation Completed after " & GetCountStep() & " Steps")
}
```

PrintLn("Simulation Completed after " & GetCountStep() & " Steps")は、コンソール画面に"Simulation Completed after "という字句、続いてGetCountStep()という関数の値、そしてさら

に"Steps"という字句を書き出すというルールです。GetCountStep()という関数は、そのときのステップ数を入手する関数ですから、この場合、終了した際のステップ数に相当します。なお、ここで用いられた、「..."」は引用符の中をそのままプリントさせることを意味し、「&」はさらにプリントさせる内容が続くことを意味しています。ここまで作業したらtachibana3(D)という名前で保存して、実行しましょう。コンソール画面にメッセージが出たでしょうか(図8.5参照)。

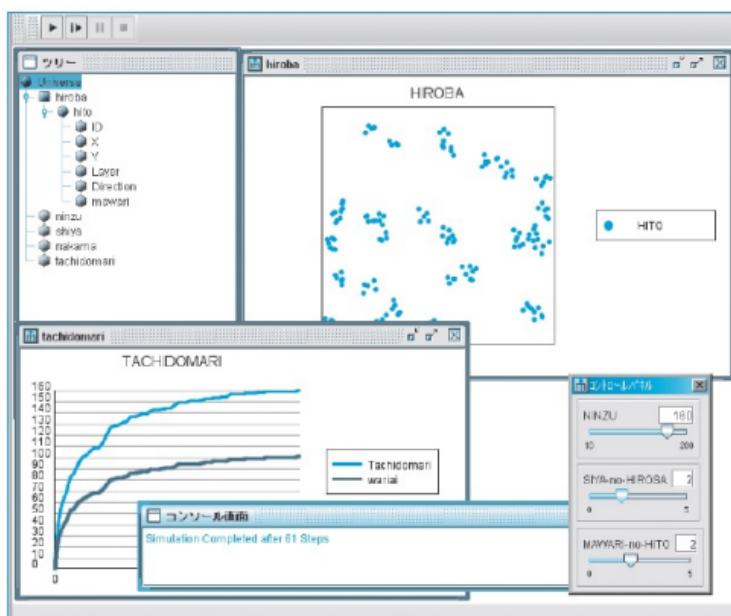


図8.5

Univ_Step_End{ }の中で、シミュレーション終了条件が満たされたときに、シミュレーションを終了させるだけでなく、上ではUniv_Finish{ }の部分でさせていたコンソール画面に書き出すルールも同時に実行させてしまうこともできます。

```
If Universe.tachidomari == Universe.ninzu Then
    ExitSimulationMsgLn("Simulation Completed after " &
```

```
GetCountStep() & " Steps")
End if
```

というルール表記です。ExitSimulationMsgLn()は、ExitSimulation()とPrintLn() の機能をひとつにまとめたものです。

新しく学んだ事項

- 時系列グラフの設定
- 集計する技法
- コンソール画面の利用
- Univ_Step_Begin{ }, Univ_Step_End{ }, Univ_Finish{ }を活用する
- ExitSimulation()
- PrintLn()
- ExitSimulationMsgLn()

練習問題

8.1

歩き回っているhitoエージェント数をこの章で作った時系列グラフに追加して出力させてみる。



Universe.ninzu – Universe.tachidomari

8.2

立ち止まっている人が全体の90パーセントを超えたたら、シミュレーションを終了させるようにルールを変

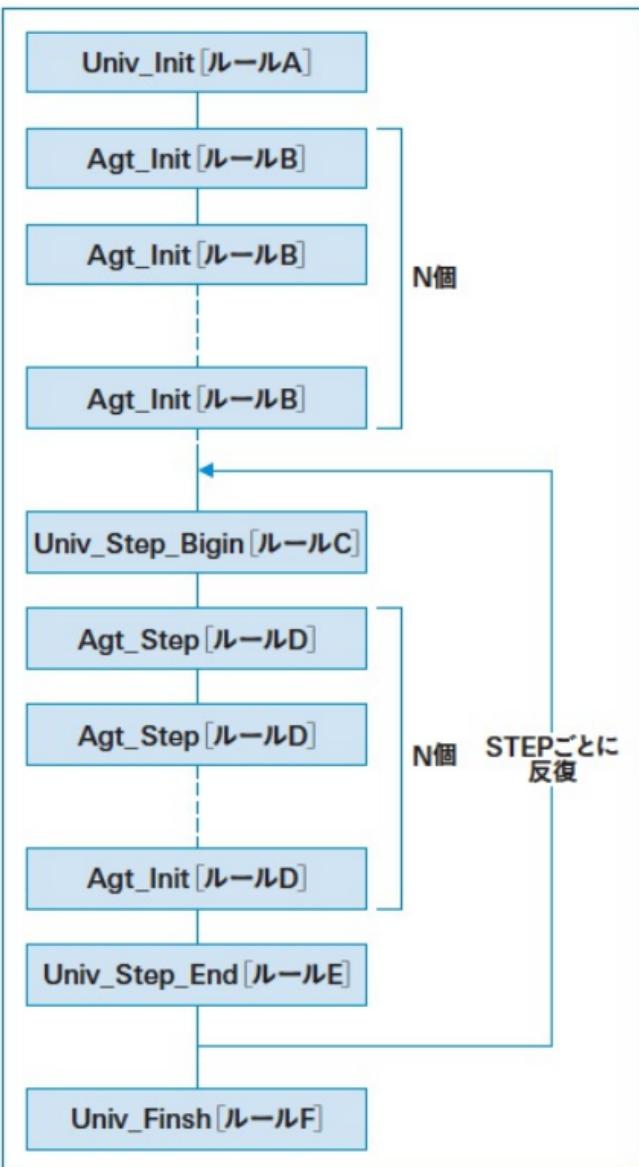
える。

Universeのルールエディタが複雑なのはなぜ?

artisocにおけるルール処理の順番について、少し詳しくまとめておきます。エージェントのルールは、Agt_Init{}のルールとAgt_Step{}のルールという2つのセクションから構成されていました。それに対し、Universeのルールは、Univ_Init{}, Univ_Step_Begin{}, Univ_Step_End{}, Univ_Finish{}の4つのセクションで構成されています。それぞれ、次のようなルールが書き込まれます。

Agt_Init{}	最初に一度だけ実行させるルール
Agt_Step{}	毎ステップ実行させるルール
Univ_Init{}	最初に一度だけ実行させるルール
Univ_Step_Begin{}	毎ステップの最初に実行させるルール
Univ_Step_End{}	毎ステップの最後に実行させるルール
Univ_Finish{}	最後に一度だけ実行させるルール

ルールの処理順序についてまとめてみましょう。エージェントが1種類で、N個いるとします。その場合のルールの処理の順序は図のようになります。



(光)

第9章

格子型空間の構造を活用する

9.0 空間に「格子」の存在を想定しましょう

- artisocの空間は、実は連続的なのです
- 格子型空間を本来の格子のようにするには工夫が必要です
- わざわざ「格子」状に空間を区切ってみます
- 「格子」があるかのようにエージェントをふるまわせましょう
- 混み合う映画館での座り方をモデル化します

9.1 ほんとうは「格子」になっていない

今まで、oozoraとかhirobaとかいう名前の空間をモデルに作ってきました。実は、これらは「格子型」と呼ばれている空間です（確認したければ、各モデルを起動して、空間のプロパティを開いて下さい。デフォルトで「格子モデル」という「空間の型」に設定されています）。しかし、今まで「空間の型」を全く意識せずに（こちらも、この点をあえて説明せずに）、空間を設定してきました。実は、artisocの格子型空間では、格子が単なる黒線の役割しかせず、エージェントは格子があることを気にしないで移動できるのです（ちなみに、もう一つの空間の型である「六角モデル」については第28章で扱います）。

まず、このことを実際にモデルで確認してから、本来の「格子型」として空間を利用する方法を学ぶことにしましょう。

空間のデフォルトの大きさは 50×50 ですが、 6×5 に設定して小さな空間で（言い換えると、大きく見える空間で）、エージェントが実際にどのように動くのかを観察しましょう。

1. Universeの下にakichiという名前で、Xが6、Yが5の空間を作ります。空間の種別は格子型（デフォ

ルト値のまま)です。

2. akichiの下にhitoエージェントを作ります。エージェント数は1にします。
3. 出力設定の作業に入ります。マップ出力(デフォルト値のまま)の設定です。ここでは今までと違う操作をします。設定画面に注目しましょう。罫線表示を選択して下さい(チェック記号が現れます)。表示型(チェス型か囲碁型を選ぶ)はチェス型をクリックして下さい。今までのように、マップ要素リストにはhitoエージェントを追加して下さい(図9.1)。

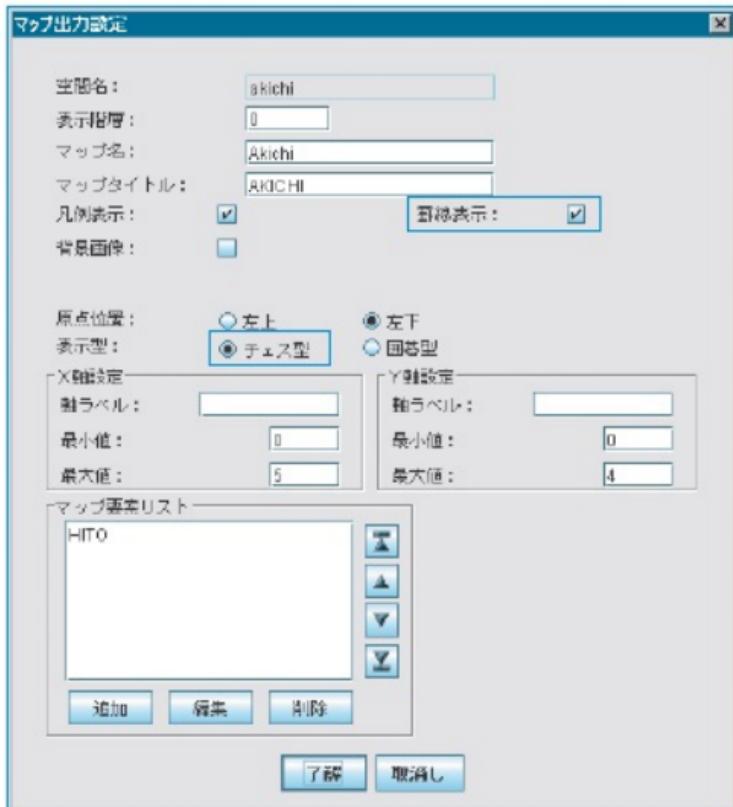


図9.1

このように出力設定をすると、空間の構造がよくわかります。このモデルを、celltestという名前で保存して下さい。実行ボタンを押して下さい。格子模様がみえるakichiの左下端の格子の枠の中にhitoが存在しているはずです。

さて、ここで画面の表示に注目して下さい。次のような重要な特徴があります（図9.2）。

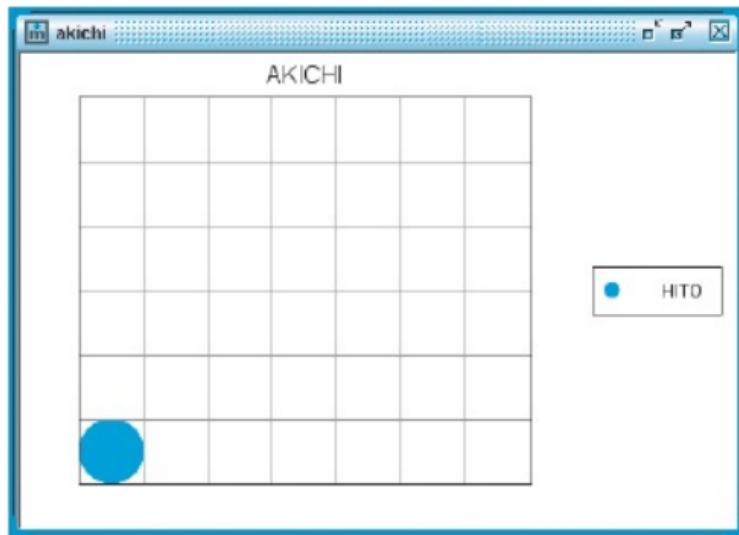


図9.2

1. hitoエージェントは、まだ何も設定していないので、位置は、X座標も0、Y座標も0のはずです（確かめましょう）。しかし一目瞭然ですが、原点（左下）ではなく、原点を左下端に持っている格子の中に位置づけられています。
2. 空間の大きさは 6×5 で設定したはずです（確かめましょう）。格子の数はいくつでしょうか？ 横に7つ、縦に6つ並んでいます。

実は、エージェントを図示するアイコンの中心はエージェントの位置ではなく、それより右に0.5、上に0.5す

れているのです。そうすると、右上端に位置するエージェントを図示する図形はどこに来るのでしょうか。エージェントは(6, 5)にあるのですから、そのアイコンは(6.5, 5.5)が中心となります。すると 6×5 で空間を表示するとアイコンは図示されなくなります。そのようなことが起こらないように、artisocでは、マップ出力を設定すると、自動的に最右列と最上行が「のりしろ」になるように、 7×6 で図示されるのです。なぜ、このように面倒なことをするのか（見かけをわざわざずらすのか）、は次節で格子型を活かすルールを学ぶときにわかります。

9.2 空間は実数的だった

エージェントを動かしてみましょう。hitoエージェントのルールエディタを開いて、

```
Agt_Init{  
    My.X = Rnd() * 6  
    My.Y = Rnd() * 5  
    My.Direction = 30  
}  
Agt_Step{  
    Forward(0.3)  
}
```

と書き込んで下さい。

上書き保存してから、「ステップ実行ボタン」を押して下さい。このボタンを押すと、1ステップだけ実行されます。エージェントがどこにいるかを確認しながら、ステップ実行ボタンを1回ずつ押して、エージェントの動きを観察して下さい（[図9.3](#)参照）。

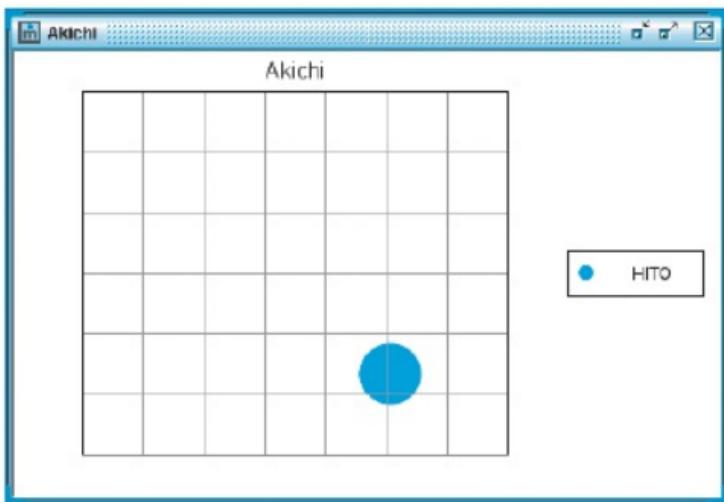


図9.3

特に次の点に注目して下さい。

1. エージェントを表すアイコンと格子の罫線とが重なることもある(エージェントが存在している位置のX座標やY座標は整数ではない)
2. 右や上の端で一瞬消えて、同時に下や左の端に現れる(空間がループしている)
3. 動き方も細かい(小数単位で毎ステップ移動できる)

このように、格子状に罫線が入っていても、エージェントの位置や動きは、格子を単位にしているのではなく、まるで実数空間のような構造をしているのです。言い換えると、格子型空間になってはいますが、格子を無視してエージェントは動き回ることができるのです。

実は、今までartisocのこのような空間の特徴を利用してモデルを作ってきました。たとえば、エージェントをRnd() * 50で空間に配置したり、Rnd() * 360でいろいろな向きにしたり、というルール記述は、空間が実数的だからエラーにならなかったのです。

artisocがこのような設定になっていることは、モデルを作る側(私たち)に大きな自由度を与えてくれています。そのために、あえて空間を「格子型」としてモデル化するときには、今までとは少し異なる特別な技法が必要になるのです。

9.3 「格子」があるかのようにエージェントを動かす

格子型の空間というのは、将棋盤(チェス・ボード)のように空間が格子状に区切られている空間です。そして格子で区切られた枠のひとつひとつがエージェントのいることができる場所になっているような構造になっています。格子型の空間では、エージェントの存在する場所がちょうど独房(セル)になっているので、セル型空間と呼ばれることもあります。格子状に区切られていても、セルではなく、碁盤のように交点のみがエージェントのいることができる場所であると考えることも可能です。具体的には、マップ出力設定画面で、罫線表示と囲碁型の2つの設定をオンにします。こうすると、エージェントは罫線の交点に表示されます。

それではいよいよ空間が格子型になっていることが意味を持つように、エージェントの動きに制約を加えることにしましょう。

まず、エージェントは格子から格子へと、とびとびにしか移動できないので、移動距離はとびとびの整数値でなくてはなりません。また、移動方向も360度どこでもよいというわけにはいきません。ある格子にいるエージェントから見ると、周囲の格子は8個ですから、移動できる方向も8通りしかありません。なお、ここでは、将棋の桂馬(チェスのナイト)のような動きは除外します([図9.4](#))。

3		2		1
	3	2	1	
4	4		0	0
	5	6	7	

图9.4

では、格子の中をとびとびに移動するエージェントを作つてみましょう。ルールを次のように変えて下さい。

```

Agt_Init{
My.X = Int(Rnd() * 6)
My.Y = Int(Rnd() * 5)
}
Agt_Step{
ForwardDirectionCell(3, 1)
}

```

最初の部分は、格子のどこかにランダムに配置するために、座標の値を整数化しています。Int()は、小数点以下を切り捨てて整数にする操作です。Rnd() * 6は0以上6未満ですから、Int(Rnd() * 6)は0から5までの整数になります。

また、シミュレーションの最初に動く方角(Direction)を決めるルールを削除しています。その代わりに、毎ステップ移動するルールが新しい表現になって、そこに動く方角(Direction)も埋め込まれています。ま

た、格子型(セル型)に対応(Cell)しています。つまり、

```
ForwardDirectionCell(3, 1)
```

という新しい関数になっています。これは、エージェントを毎ステップ、左斜め上方向([図9.4](#)の「3」方向)に「1」枠分だけ動かす関数です。

では、realcellという新しい名前を付けて保存してから「ステップ実行」して下さい。

エージェントの動きを確認しましょう。これで、格子型空間の上を格子の中だけ動き回るエージェントの特徴と、格子型空間の特徴がつかめたことだと思います。ForwardDirectionCell()関数の最初の数値(現在は3)を他の数値(7以下の整数)に変えて、エージェントの動き方がどのように変わるかを、[図9.4](#)と比べながら、確認して下さい。

このようにエージェントを格子があるかのように動かすと、エージェントは決してマップakichiの「のりしろ」部分に表示されないことがわかると思います。それでは、マップを編集してのりしろをなくして下さい。これは、「マップ出力設定」ウィンドウのX軸設定、Y軸設定の最大値をデフォルト値より1少ない値に設定し直す作業です。

作業が終ったら、realcell(B)という名前を付けて保存して、再び「ステップ実行」して下さい。格子の状態が変わったことを確認しましょう。6×5になっているはずです。

[9.4](#) 格子を利用するモデルの外枠を作る

これから、格子型空間の構造を活用したエージェントの動かし方を学びましょう。

ある劇場に 10×20 の椅子があり、そこにモデルの外から操作するn人が観劇に訪れるが、周囲に人がいて混んでくると空いている席に移動する。

まず、モデルを作る準備作業です。

1. Universeの下にgekijo空間を10×20の大きさで設定する(なお、ここでは「ループしない」を選択)
2. Universeの下に整数型変数kankyakuを作る
3. gekijoの下にhitoを作る(エージェント数は0)
4. マップ出力を設定する(墨線表示をチェックして、チエス型を選択する。そして、「のりしろ」がかくれるように、空間のサイズを9×19に変える)([図9.5](#)参照)

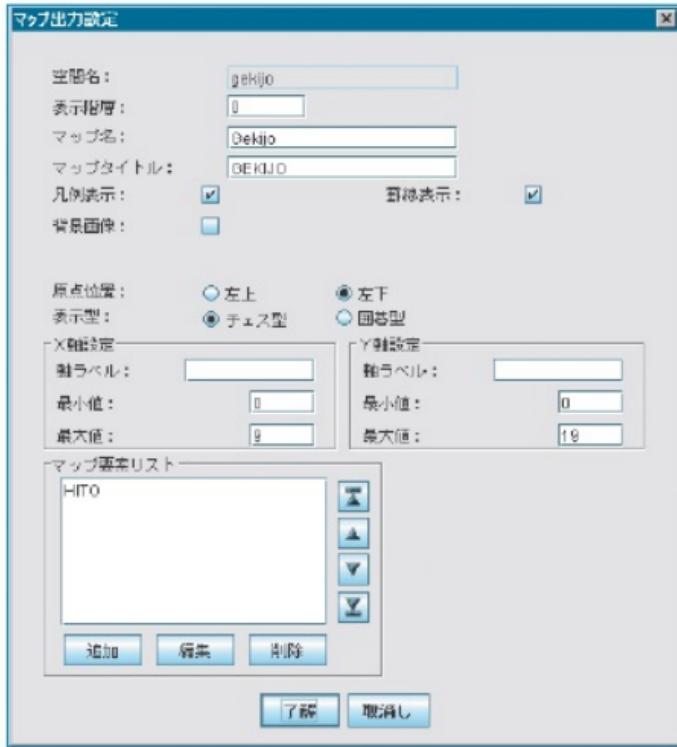


図9.5

5. kankyakuを10人から160人までの範囲で変えられるようにコントロールパネルを設定する(目盛り間

隔は10)

以上の作業を終えたら、eigakanという名前で保存して下さい。

9.5 格子を意識したルール表現を使う

では、ルールの記述作業に移りましょう。

まず、Univ_Init{}にkankyakuの数だけhitoエージェントを生成させて、gekijoにランダムに座らせるルールを書きます。

```
Univ Init{
Dim i As Integer
Dim hitobito As Agtset
For i = 0 To Universe.kankyaku - 1
    CreateAgt(Universe.gekijo.hito)
Next i
MakeAgtsetSpace(hitobito, Universe.gekijo)
RandomPutAgtsetCell(hitobito, False)
}
```

上のルールで、RandomPutAgtset()の「格子(セル)」版であるRandomPutAgtsetCell()を使って、格子に区切った空間にエージェントをランダム配置しています。ただし、括弧の中の表記が違っていて、Falseが追加されています。これは、1つの格子枠内に複数のエージェントを配置できないようにする場合です。このモデルでは格子(セル)は座席ですから、1つの座席に複数のhitoが座ることのないように設定する必要があります(もし重複を許すなら、RandomPutAgtsetCell(hitobito, True)となります)。また、上でMakeAgtsetSpace()ではなく、MakeAgtset()を使うことも可能ですが、ただし、第7章で学んだように、括弧の中の表記が変わってくることに注意して下さい。

次に、エージェント(hito)の動くルールを書き込む作業です。エージェントは周囲を見回して、視野1の範囲に4人以上他人がいると窮屈だと感じて、視野3の範囲で空席をさがして、空席があればそこに移ります。

```
Agt_Step{
```

```
Dim mawari As Agtset
MakeAllAgtsetAroundOwnCell(mawari, 1, False)
If CountAgtset(mawari) >= 4 Then
    MoveToSpaceOwnCell(3)
End if
}
```

上のルールは既に第6章で学んだルール表現(周りの様子を認識し、状態に応じて行動を変える)の応用です。この中で、`MakeAllAgtsetAroundOwnCell(mawari, 1, False)`は、既に使ったことのある`MakeAllAgtsetAroundOwn()`の「格子(セル)」版です。しかし、格子に区切られている空間では、視野の意味が異なるので、後述します。

`MoveToSpaceOwnCell(3)`は初めて出てきた関数です。エージェントを格子(セル)があるように動かすときにも重要なルールに対応しています。エージェントを実数的に動かす場合には、狭い範囲に(たとえば1点の上に)多数のエージェントが存在できます(第3章の、`tobutori`モデルの準備作業を思い出して下さい)。しかし、格子上を動かす場合には、エージェントはセル(または格子の交点)にしか存在できません。そこで、空いている格子をさがすというルールが重要になるのです。`MoveToSpaceOwnCell(3)`は格子型の視野「3」の範囲で、空いている格子があれば、そこに移るというルールです。

さて、空間を格子状であるとみなしたときの「視野」は、今までの視野の捉え方と少し異なります。[図9.6](#)に、その違いを図示してあります。

2	2	2	2	2	
2	1	1	1	2	
2	1	0	1	2	
2	1	1	1	2	
2	2	2	2	2	

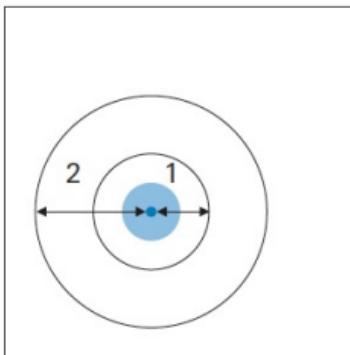


図9.6

上のルールでは、自分の周囲(視野が「1」に設定されているので、隣接している8つの格子)に4人以上いると、混みすぎだと感じて、視野3の範囲で(周囲48席のなかで)空いている席に移動します。このモデルを上書き保存してから、実行しましょう(図9.7参照)。観客数が少なければ、すぐに移動するエージェントがいなくなりますが、多いとなかなか落ち着けないことが観察できるはずです。

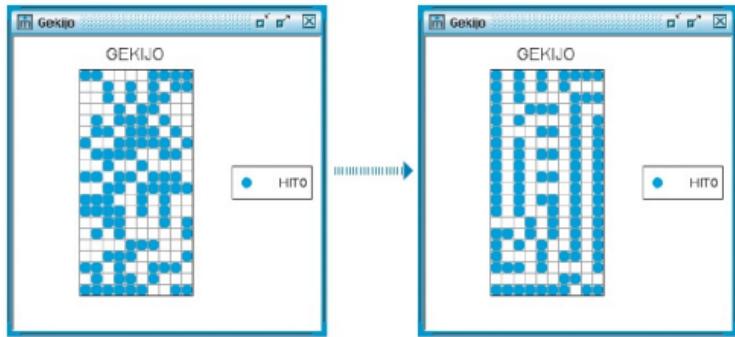


図9.7

新しく学んだ事項

- Int()
- ForwardDirectionCell()
- RandomPutAgtsetCell()
- MakeAllAgtsetAroundOwnCell()
- MoveToSpaceOwnCell()
- 格子(セル)型空間の方向の表し方
- マップ出力設定による罫線表示の方法(チェス盤と囲碁盤)
- マップ出力設定による空間の「のりしろ」の隠し方
- 「視野」の捉え方の違い

練習問題

格子に区切られている空間を想定したエージェントの動かし方に慣れましょう。

9.1

この章で作ったrealcell(B)モデルで、エージェントは毎ステップ、ランダムな方向に動く。



定石のRnd() * 360は使えません。

9.2

まず、realcell(B)モデルで罫線を囲碁盤のようにマップ出力するように設定し直して下さい。出力設定の編集機能を使います。その上で、次の課題をやって下さい。

エージェントは毎ステップ、ランダムに上下左右に(つまり罫線に沿って)1目しか動けない。



条件分岐が複雑です。[図9.4](#)は格子型モデルを作る上でとても大事です。

見方を変えると周りも変わる

まずは準備です。東南アジア大陸部の地図を見ながら、どの国がどの国と国境を接しているか調べましょう。国土という面と国境という境界線からなる普通の地図から、国土を点に、国境を接している国どうしを線で結ぶことにより国境の接し具合を表した地図を作ることができます(図1)。これを「双対地図」といいます。このように、「面」の世界と「点と線」の世界とを対応させることができます。

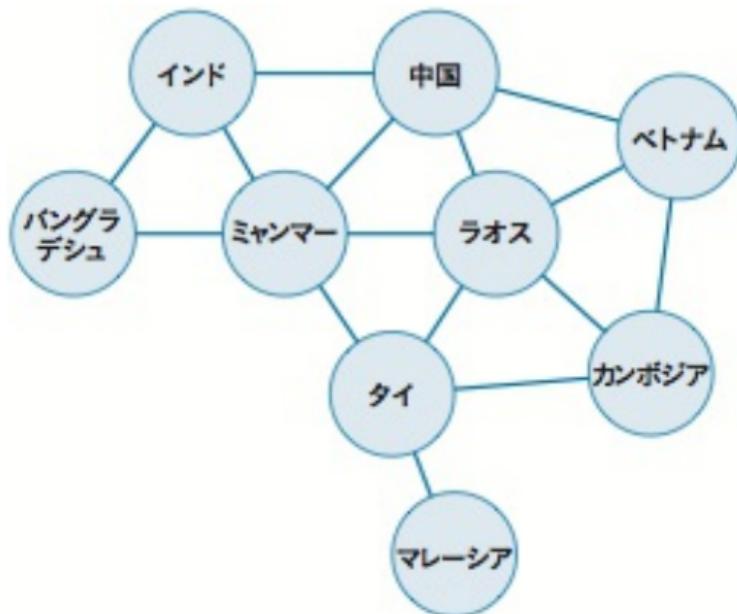


図1

ここからが本題です。

正六角形のタイルを敷き詰めた床があります。個々のタイルの周囲にはいくつタイルがあるでしょうか

([図2](#))。答えは6つですね。平面のこのよう区切り方を「六角」モデルとか「蜂の巣」モデルとか言います。ここで、接している状態を上の双対地図のように表してみましょう([図3](#))。正三角形が並んでいますね。もちろん、頂点がタイル(面)に対応しています。各頂点から辺が6つ出ているので、各点は6つの点と繋がっています。これで、「蜂の巣」モデルと「三角格子」モデルが、ある意味で同じになることがわかりますね。

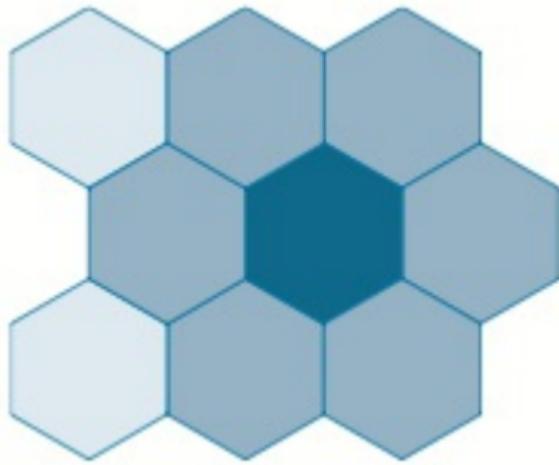


図2

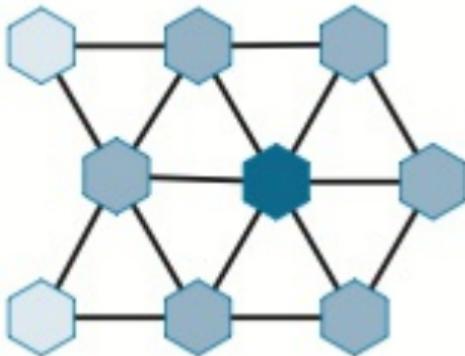


図3

正方形のタイルを敷き詰めた床があります（[図4](#)）。個々のタイルの周囲にはいくつタイルがあるでしょうか。六角形タイルの場合ほど簡単ではありません。まず、面と面とがくっついている所だけに注目すると4つで、普通の「四角格子」になります（[図5](#)）。しかし、頂点だけが辛うじて接している所も勘定に入れると8つです（[図6](#)）。周りを4つだとする見方を「ノイマン近傍」、8つだとする見方を「ムーア近傍」と呼びます。

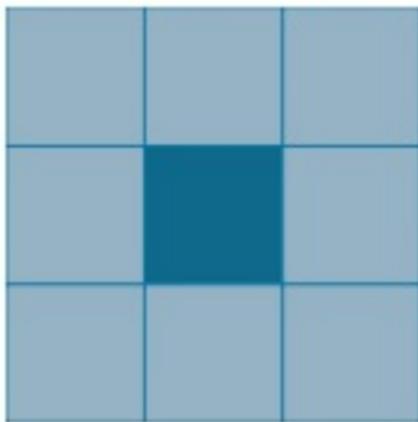


図4

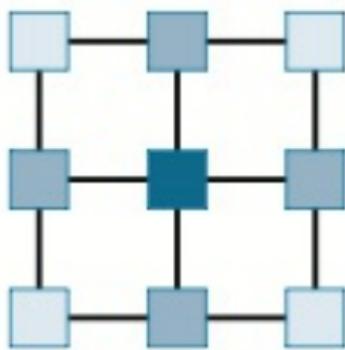


図5

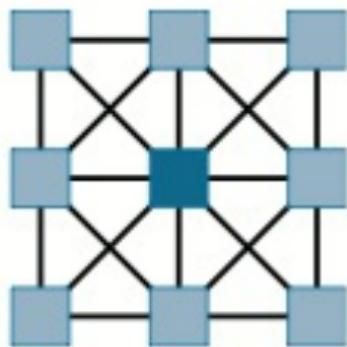


図6

周囲との関係をどのように見るのは、学問分野によって違うようですが、artisocでの捉え方の基本（デフォルト）としては、実は、普通の座標系のような見方を採用しています。それをタイルが敷き詰められた床に見立てる場合には、正方形のタイルを想定していますが、「ムーア近傍」を採用しているので、「四角格子」とは呼ばずに、単に「格子モデル」と名付けています。「ノイマン近傍」の見方でモデルを作る場合には気をつけて下さい。少し技巧を要します。

なぜartisocはムーア近傍の捉え方を採用したのか、ですって。あまり深く考えないで下さい。まあ、セルオートマトンの考え方に基づいたライフゲームのモデルを作るときに便利だから、といった気楽な理由です。ちなみに、マルチエージェント・シミュレーションの古典であるシェリングの分居モデルも、ライフゲームも、周囲の捉え方として「ムーア近傍」を採用しています。

もちろん「蜂の巣」ないし「三角格子」モデルの見方もできるようになっています。こちらは「六角モデル」と名付けています。

なお、artisocでは格子モデルでも六角モデルでも空間をどのように区切っているかを、視覚的に表した

いときには、チェス盤型(つまり罫線で区切られたタイルにエージェントが存在する)と囲碁盤型(つまり罫線の交点にエージェントが存在する)とを選ぶことができます。チェス盤型だとセルオートマトンを敷き詰めたイメージになり、囲碁盤だと格子(点と線)あるいはネットワーク(ノードと枝)のイメージになります。

もっとも、現実の世界は、四角いタイルでできているわけでも六角タイルでできているわけでもないので、しょせんは便宜のレベルです。繰り返しになりますが、artisocのデフォルトは、格子に区切った空間ではなく、実数的な2次元座標系になっています。したがって、エージェントは格子に拘束される必要がなく、自由に動き回れます。また、エージェントの間のリンクも距離に関係なく自由に結べます。もちろん、格子の拘束を受けるように設定することも可能です。artisocは自由度が大きいのです(えへん)。

第10章

第1部の修了を記念して： シェリングの「分居モデル」を作る

10.0 artisocの有り難さをじっくりと味わいましょう

- artisocの基本を一通り学びました
- この章では新しく学ぶ技法は全然ありません
- 格子型に区切られた空間をエージェントが動き回り、やがて落ち着きます
- アイデアをモデルに結びつけるところが肝腎です
- モデルの改良も簡単です

10.1 分居モデルとは

2005年ノーベル経済学賞を受賞したトマス・シェリングは、おそらく初めてマルチエージェント・シミュレーションの手法を社会科学に取り入れた研究者です。彼は、アメリカの都市で人々が民族集団毎に分かれて生活する現象に注目しました。通説によると差別意識や相互排他意識が地域社会の分居を促すとされていましたが、シェリングはこの通説を疑問視して、シミュレーションによる研究をしたのです。その結果、個々人の意識がさほど排他的でなくても結果として地域社会の分居が生じてしまうことを鮮やかに示したのです。簡単なルールで社会現象の本質を捉え、個々人の意思とその相互作用で生まれる社会現象との直感に反する関連を見事に明らかにした古典といえます。

もっともシェリングが研究を行なった1960年代には、まだ便利なコンピュータはなく、彼が使ったのは、23枚の1セント硬貨(ペニーと呼ばれる銅色のコイン)、22枚の10セント硬貨(ダイムと呼ばれる銀色コイン)、チェス盤(正確にはチェック盤)、そしてサイコロでした。つまり、64軒の住宅が 8×8 に並んでいる地域社会に銅色人種の居住者が23、銀色人種の居住者が22住んでいる状況をシェリングはモデル化したわけです。

	#		#	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				#		

図10.1

シェリングのモデルでは、個々の居住者は、近隣住民に占める同色居住者の比率が一定限度（これを専門用語では「閾値」といいます）以上ならば満足して、そこに住み続けるが、そうでなければ空き地（空き家）に引っ越しすという行動をとります。ここで、近隣とは、その家の東西南北4軒だけでなく、北東・南東・北西・南西の4軒も含むことにします。シミュレーションを簡単にするために、シェリングは全ての居住者の許容限度（閾値）を1/3に統一しました。つまり、この比率よりも大きければ、各住民は満足しているので、引っ越ししません。

まずサイコロを振って、最初45のコインがある場所をランダムに決めました。そして、個々のコインについて満足か不満かを計算し、不満なコインを周囲の空き地の中からサイコロを振って決めた場所へと移す作業を不満なコインがなくなるまで繰り返しました。そして、最終状態の各コインにとっての近隣の同種コインの比率を調べたのです。

もし全ての住民が、近隣住民のうち同色人が1/3以上ならば満足するならば、この地域社会全体では、どの程度人々が分かれて住むことになるのでしょうか？住民はかなり寛容（同色人の割合が半分以下でも満足）なのだから、社会はあまり分居しないだろう、と事前に予想できるのではないでしょうか。しかし、シェリングは、この手間のかかるシミュレーションを繰り返すことによって、個々の居住者の許容限度が

低くとも、地域社会全体では分居が進むことを示したのです。

10.2 コンピュータのなかの分居モデル

シェリングはコイン、サイコロと本物のチェス盤を使いました。わたしたちは、artisocを使いましょう。

住宅が 8×8 の状態に立ち並んでいる地域社会に、銅色人種の住民が23家族と銀色人種の住民が22家族と住んでいる。兩人種とも異人種に寛容で、同人種が近隣住民のうち $1/3$ 以上ならば満足して住み続ける。そうでない場合に限り、近所の空き家に引っ越しする。

まず、例によって準備作業から入りましょう。もうartisocに慣れたことと思いますので、詳しい具体的な作業の説明は省略します。

1. Universeの下にchessboardという名前の空間(ループしない)を 8×8 で作る
2. chessboardの下にpennyを0個、dimeを0個作る(エージェントはUniv_Init{ }で生成させます)
3. 出力設定の作業として、chessboardのマップ出力(デフォルトではなく、X座標Y座標の最大値を各々7、罫線あり、チェス型に設定)を設定する。要素設定のとき、エージェントの色を本物のコインの色に似せるのも一興ですね(印刷の都合上、pennyは青っぽく見えます)
4. schellingという名前で保存する
5. 実行ボタンを押して、設定が正常か確認する(停止ボタンを押すのを忘れずに)([図10.2](#)参照)

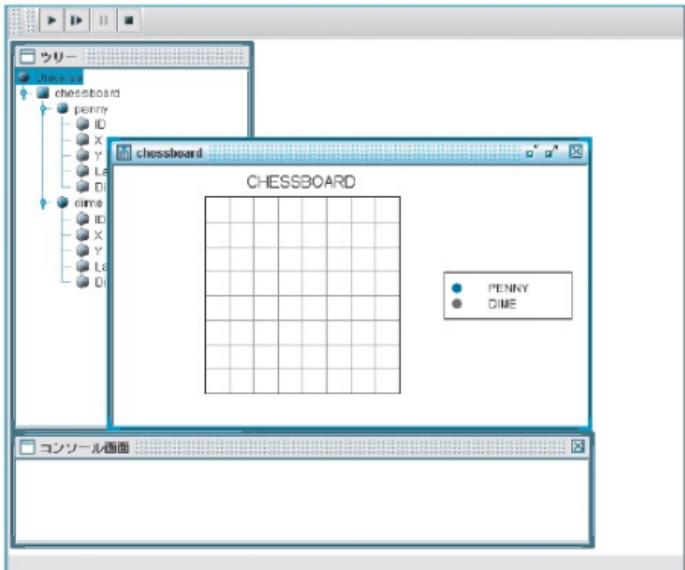


図10.2

次にルールを書き込む作業に入りましょう。

1. コインをチェス盤にランダムに置くルールを書き込む

```
Univ_Init{
    Dim i As Integer
    Dim coin As Agtset
    For i = 0 To 22
        CreateAgt(Universe.chessboard.penny)
    Next i
    For i = 0 To 21
        CreateAgt(Universe.chessboard.dime)
    
```

```
Next i  
MakeAgtsetSpace(coin, Universe.chessboard)  
RandomPutAgtsetCell(coin, False)  
}
```

2. 1セント硬貨(penny)のルールを書き込む

周囲の状況を認識し、満足かどうか判断させます。不満な場合には、空き家を探して移動します。

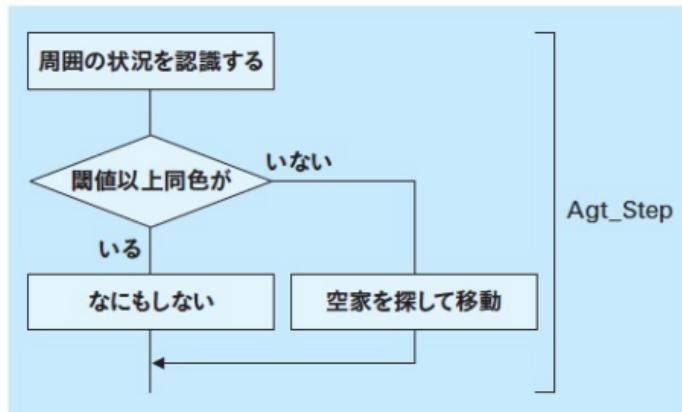


図10.3

```
Agt_Step{  
Dim neighbor As Agtset  
Dim NP As Integer  
Dim ND As Integer  
Dim rate As Double
```

```

MakeOneAgtsetAroundOwnCell(neighbor, 1,
Universe.chessboard.penny, False)

NP = CountAgtset(neighbor)
MakeOneAgtsetAroundOwnCell(neighbor, 1,
Universe.chessboard.dime, False)
ND = CountAgtset(neighbor)
If NP + ND == 0 Then
    rate = 0
Else
    rate = NP / (NP + ND)
End if
If rate < 1/3 Then
    MoveToSpaceOwnCell(2)
End if
}

```

上のルールで、rateを計算させる部分でゼロで割ることをしない（エラーにならないように）工夫をしていることに注意して下さい。

3. 10セント硬貨(dime)についても同様のルールを書き込む
pennyのルールと、下の1カ所だけ違います。

```
rate = ND / (NP + ND)
```

あとは同じです。そこで、pennyのルールをdimeのルールエディタに「コピー・アンド・ペースト」して、必要箇所を修正することも一法です。

これで完了です。上書き保存しましょう。実行ボタンを押して、サイコロを振る手間が無くなった幸福を

味わって下さい。

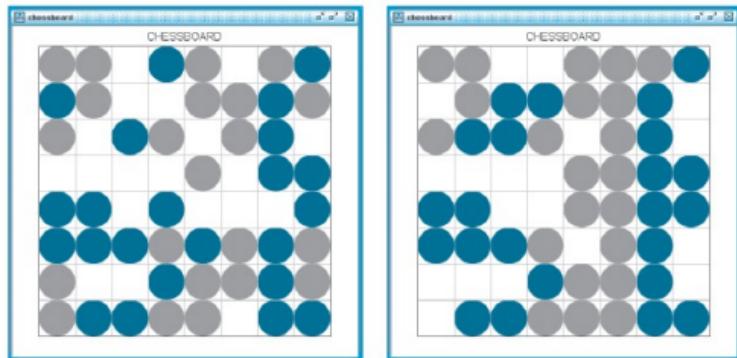


図10.4

10.3 若干の補足

以上のモデルは、実は厳密に言うと、シェリングのオリジナルなモデルと不満な場合の移動方法が異なっています。シェリングはもともと、不満なコインはチェス盤全体のどこかの空き地に移動するルールを用いています。ここでは、視野「2」で空き地を探すようにルール化しました。チェス盤全体を見回せるように広い視野を設定すると、すぐにどこかの空き家に引っ越して、全てのコインが満足してしまいます。

シェリングは、コインが空き地を探し回るプロセスではなく、結局分居してしまうという結果を分析したかったので、視野の問題は重要ではありませんでした。

artisocでは、結果を求めることが大事ですが、過程を観察できるというメリットがあります。そこで、とりあえず、視野を「2」に設定してみました。

10.4 シェリングの分析を再現する

では、シミュレーションの過程を出力マップで観察するだけではなく、シェリングが行なった分析を再現し

ましょう。それは、地域社会の全体的な分居の程度がどのように変化したか、です。

個々のエージェントにとっての分居度ではなく、全体の分居度を時系列グラフで表す。

1. 集計用の変数(実数型)averagelevelをUniverseに追加する(averageとlevelの間にはスペースを置かず、くっつけて書き込みます)
2. 毎ステップ初期化する

```
Univ_Step_Begin{
    Universe.averagelevel = 0
}
```

3. 各ステップ、個々のエージェントの分居度を集計する

ルール表記は下のようになりますが、どこに挿入すれば正しく集計されるのか考えて下さい。

```
    Universe.averagelevel = Universe.averagelevel + rate
```

4. 時系列グラフの出力設定をする(Y軸設定の最大値を1に、目盛り間隔を0.1にする)

出力値は、Universe.averagelevelそのままではなく、

```
    Universe.averagelevel / 45
```

です。これで平均値が求まります。

schelling(B)という名前を付けて保存してから実行しましょう。全体的な分居度がどんどん上がっていく様子が見て取れると思います。

満足しているエージェントの割合を時系列グラフに表示する(schelling(C))。

1. 集計用の変数(実数型)satisfiedをUniverseに追加する
2. 毎ステップ初期化する
3. 各ステップ、個々のエージェントについて、満足していれば、カウントする

ルール表記は下のようになりますが、どこに挿入すれば正しく集計されるのか考えて下さい。

```
Universe.satisfied = Universe.satisfied + 1
```

4. 時系列グラフの出力設定をする

出力値は、Universe.satisfiedではなく、

```
Universe.satisfied / 45
```

です。

全員が満足したら終了する(schelling(D))。

シミュレーションの終了条件を指定するルールを書き込みます。全員が満足すると終了させることにしましょう

```
Univ Step_End{
If Universe.satisfied == 45 Then
  ExitSimulationMsgLn("Every one satisfied after " &
GetCountStep() & " steps")
End if
}
```

もし、不満な人が残っていても50ステップたつたら終了させるには、下記のようなルールを追加すればよいでしょう。

```
If GetCountStep() > 50 Then
  ExitSimulationMsgLn("Not all satisfied until 50 steps")
End if
```

練習問題

シェーリングのオリジナルな分居モデルを作つてみると、いろいろなアイデアが出てきます。オリジナルなモデルを発展させてみましょう。

10.1

次のような工夫をすれば、第2章で体験した分居モデル(02segregation)が再現できます。

- 空間を広くしてエージェントを増やす
空間を大きくしてエージェント数を大きくすると、実行速度が遅くなります。見づらくならない程度にしておきましょう。
- エージェント数をコントロールパネルで操作できるようにする
エージェント数が格子数以上にならないよう、注意しましょう。
- 視野をコントロールパネルで操作できるようにする。

10.2

さらには、もっと「すぐれた」モデルを作ることも可能になるはずです。

- エージェントの種類を2から3にする
- エージェント種が異なるとルールも異なる

変数には必ず「型」があります。「変数の型」はそれぞれの変数がどんなタイプ(型)の値をとるのかを前もって決めたものと考えて下さい。たとえば、整数型の変数は、-3, -2, -1, 0, 1, 2といった整数の値をとります。実数型の変数は、3.1415...とか2.2362...といった実数の値をとります。もっとも、実際にはコンピュータは本当の実数の値を取り扱うことはできないので、小数点以下が何桁もある数値(擬似的な実数)ということになります。

整数型の変数に実数値を代入すると、小数点以下が切り捨てられて整数値として代入されます(たとえば、3.14なら3)。また逆に、実数型の変数に整数値を代入すると、値は変わりませんが実数値として代入されます(たとえば、5なら5.0)。注意して下さい。

変数のとる値は数字だけとは限りません。たとえば、文字列型の変数は、YESとかYAMAKAGEといった文字列を値としてとります。そして、artisocには、エージェント型およびエージェント集合型という変数の型もあります。エージェント型とは個々のエージェントをその値としてとります。エージェント集合型はエージェントの集合をその値としてとります。

そのほかにも、ブール型、長整数型、エージェント種別型、空間型という変数の型があります。とりあえず今の段階では必要はないので、このような型があるということだけ覚えておきましょう。これらの変数の型は、ツリーに変数を作るときは、変数のプロパティ・ウインドウで設定します。ルールエディタ内で変数を作るときは、ルールとして型宣言を行ないます。型宣言のやりかたについては、第7章で説明されています。宣言の際には、整数型はInteger、実数型はDouble、文字列型はString、エージェント型はAgt、エージェント集合型はAgtset、を用います。

(光)

第2部

人工社会の発想と技法に慣れる

第1部では、マルチエージェント・シミュレーションを構想してから実行するまでの流れを、artisocを使って一通り学びました。この第2部では、エージェントどうしの相互作用をさせる基本技法を学びます。さまざまな相互作用について、artisocではどのようなルールで表現するかを説明しましょう。基本技法のレパートリーは、第2部でほぼ全て登場します。

なお、第2部では、第1部で学んだことについては適宜省略して、モデルの作り方を説明します。また、第2部の後ろの方の章では、第1部で学んだ技法だけでなく、第2部の前の方の章で初めて登場する技法も用いるケースがあります。その意味で、第2部も章の順番通りに学んでいくことを勧めます。

第11章

状況に応じた行動の選択肢を増やす

11.0 イフ文の使い方をマスターしよう

- 複雑な場合分けをルール化するにはイフ文の構造を複雑にします
- フローチャートをきちんと書くことが重要です
- ルールを見やすく書くことも大事です
- 混雑する駅で人の流れがスムーズになる現象をモデル化しましょう
- 周囲の状況を調べる技巧的な方法をこっそり伝授します

11.1 複雑な場合分け

エージェントに複雑な行動をとらせるための基本は「場合分け」です。「条件分岐」とも言います。エージェントが自分の周囲の状況や自分自身の内部状態に応じてさまざまな行動を選び分けることが可能になって、初めて「エージェントは複雑な行動をとることができる」と言えます。イフ文を用いた場合分けの基礎は、第5章で学びましたが、この章ではイフ文を複雑にして、複雑な場合分けをルールに記述する技法を説明します。この技法により、エージェントの行動の選択肢を増やす方法を習得できます。

複雑な行動をルール化するための「場合分け」ないし「条件分岐」は一般的に図11.1のようなフローチャートに描くことが可能です。

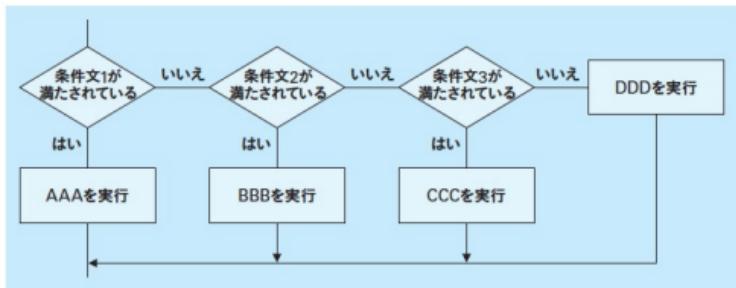


図11.1

artisocのルール表記では、下のようなフローチャートは次のような「イフ文」になります。

```

If 条件式1 Then
  AAA
Elseif 条件式2 Then
  BBB
Elseif 条件式3 Then
  CCC
Else
  DDD
End if

```

条件式1が成立していれば、AAAを実行して、End ifに飛びます。条件式1が成立していないくて、かつ、条件式2が成立していれば、BBBを実行してEnd ifに飛びます。条件式1と2が成立していないくて、かつ、条件式3が成立していれば、CCCを実行してEnd ifに飛びます。もし、条件式1, 2, 3 が全部成立しない場合には、DDDが実行されます。要するに、条件式を上から順番にチェックしていくて、該当するところのルールが実行されるのです。したがって、もし条件式1が成立していないく、条件式2も3も成立している場合にはBBBのみが実行されCCCは実行されません。

なお、「Elseif 条件式 Then」のセットは、もっとたくさんになっても構いません。また、AAA, BBBなどの実行ルールは、それ自体がイフ文で、「入れ子」構造になっていてもかまいません（イフ文の入れ子構造も、既に第5章で登場しています）。そのような「入れ子」構造が何重にもなるルールもあるでしょう。

また、条件式自体が複雑になる場合もあります。条件PとQの両方が成立している場合、どちらかが成立している場合など、論理演算子を使って複雑化できます。論理演算子については、コラム「[大小関係を表す演算子](#)、[論理関係を表す演算子](#)」を参照して下さい。

11.2 ターミナル駅の通勤客の流れ

それでは、複雑なイフ文を用いるモデルを作りましょう。

大きな乗換駅では、特にラッシュアワーに、乗降客が互いに別の電車に乗るために混雑する状況になります。そのようなとき、不思議なことに、互いにすれ違う流れが生じて、スムーズに歩ける場合があります。各人はどのように判断し、行動しているのでしょうか。果たして、スムーズな人の流れはひとりでにできるでしょうか。

[図11.2](#)のように、乗換駅のコンコースには、W線の改札口が西側に、E線の改札口が東側にあります。W線の降車客全員がE線の改札口に向かい、E線の降車客全員がW線の改札口に向かう状況を仮定します。ひとりひとりの乗降客がエージェントです。

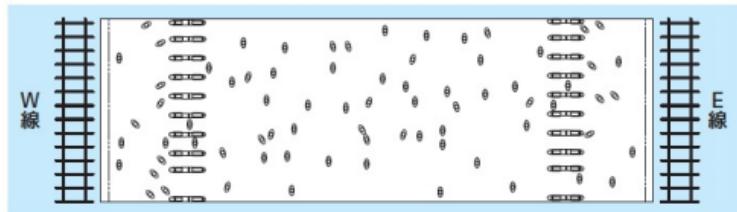


図11.2

各人は、反対方向から来る乗降客となるべくぶつからないようにするのと同時に、なるべく早く目的の改札口に向かおうとする。

エージェントの行動ルールは次のようにしてみましょう。

- 正面に対向客がいなければ、まっすぐ、速いペースで進む
- 正面には対向客がいるが、左前方か右前方にいなければ、いない方に速いペースで進む(どちらにもいなければ、ランダムに進む方向を決める)
- 正面も左右前方にも人がいる場合、一番すいている方向に進む
- 前方がどこも同じように対向客で混んでいるなら、うろうろしながら前に進む

このような基本的なルールで乗り換える乗降客の集団的行動をartisocでモデル化しましょう。

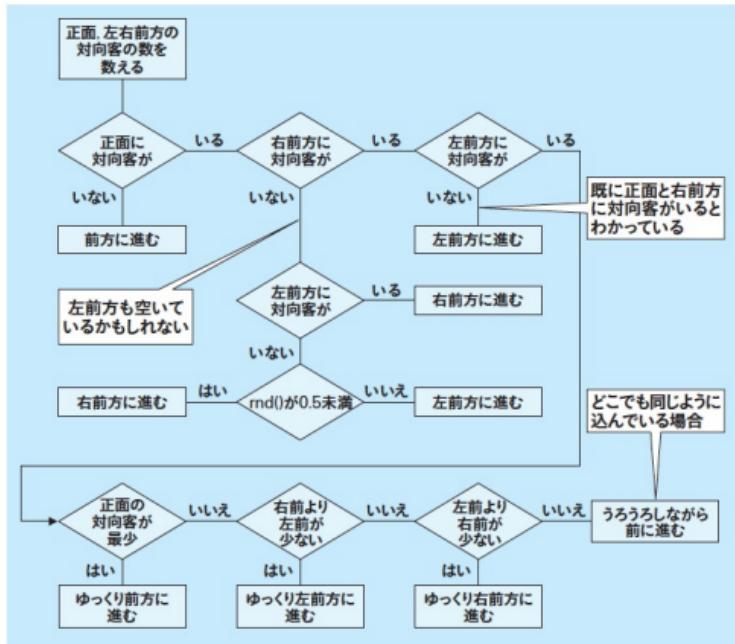


図11.3

- 前方に對向客が何人いるか調べる:ここで技巧的な周囲の調べ方を学びましょう。一般的な技法なので、さまざまな場面で応用できます。基本的には次のようにルールを複合しま

す。調べたい方向を向く → 調べたい範囲の中心に前進する → 周囲を調べる → 元の位置に戻す

で完了です。つまり、artisocの関数を、Turn(), Forward(), MakeOneAgtsetAroundOwn(), CountAgtset(), Forward(), Turn()うまく組み合わせるので。このモデルでは、最初のTurn(), Forward()で好きな方角の前方に行って調べ、最後のForward(), Turn()で元の位置に戻ります。正面ならもちろんTurn()は不要です。1ステップの間に行って戻る行動をとっているので、結果として動いてはおらず、前方を調べたことになるのです。やや技巧的な手法ですが便利なのでマスターして下さい。

- 状況に合わせて、進む方向と速さを設定する: Turn(), Forward()を使います。
- 駅コンコースの全体的な様子を把握する: 全体的状況を表す指標として、乗降客の速さの平均を調べることにしましょう。ただし、歩く速さそのものではなく、向かっている改札口の方向に近づく速さ(つまり、この場合は、東西方向の移動速度)を求めます。

11.3 モデルの枠組みを作る

ではモデル作りに着手しましょう。現実では、改札口からどんどん客が現れて、反対の各札口へ消えていくわけですが、そのような設定はまだ学んでいません(第16章で学びます)。ここでは、駅通路に乗降客が最初ランダムに配置され、彼らがどんどんと東西に歩き続ける(つまり空間はループしている)ように作ります。まず、外枠作りとシミュレーションの準備作業に入ります。

- Universeの下に空間terminalを 20×20 の大きさで作る(「ループする」(デフォルト)に設定しておいてください)
- Universeの下に、各改札口からの乗降客数を表す整数型変数peopleと駅方向の速さの平均を求めるための実数型変数advanceを追加する
- terminalの下にeastwardとwestwardの2種類のエージェント種を作る(エージェント数はデフォルト値(0)のままで)
- 2種類のエージェントの下に目的改札口方向への移動速度を表す実数型変数advanceを追加する
- peopleを10から200まで変化させられるコントロールパネルを設定する(スライドバーを選び、目盛り

間隔を10にしてください)

6. terminalと2種類のエージェントをマップ出力するように設定する
7. 平均速度を時系列グラフに示すように出力設定する(Y軸を、最小値0.0、最大値1.2、目盛り間隔0.2)設定してください。X軸については、目盛り間隔を10にします。出力値はUniverse.advance / (2 * Universe.people)とします)
8. このモデルをrush-hourという名前で保存する

なお、時々、「上書き保存」や別の新しい「名前を付けて保存」をすることを忘れていないか確かめて下さい。特に実行ボタンを押す前には!

11.4 ルールを書き込む(その1)

では、ルールを書き込みましょう。まず、シミュレーションの全体を管理するために、Universeのルールを書き込みます。

1. peopleの数だけ東向きと西向きのエージェントを生成して、ターミナル駅通路にランダムに配置します。この技法は、第7章で学んだことそのままで。

```
Univ_Init{  
    Dim i As Integer  
    Dim commuters As Agtset  
    For i = 0 To Universe.people - 1  
        CreateAgt(Universe.terminal.eastward)  
        CreateAgt(Universe.terminal.westward)  
    Next i  
    MakeAgtsetSpace(commuters, Universe.terminal)  
    RandomPutAgtset(commuters)  
}
```

2. 集計値を初期化します。

```
Univ_Step_Begin{
    Universe.advance = 0
}
```

3. Univ_Step_End{ }で、スムーズな人の流れが発生したらシミュレーションを終了させます。このセクションで、全エージェントの速さを足し合わせることもできるのですが、後述するようにエージェントのルールエディタの中で便宜的に集計しています(この便宜的手法は、第8章で登場しました)。最高速度は1.2ですから、平均が1.19になつたら終了させるルールにしましょう。

```
Univ_Step_End{
    If Universe.advance / (2 * Universe.people) >= 1.19 Then
        ExitSimulationMsgLn("Simulation completed after" &
            GetCountStep() & "steps")
    End if
}
```

とりあえず、このあたりで上書き保存して下さい。実行ボタンを押してみましょう。ここまで正しくできていると、[図11.4](#)のような画面がでてくるはずです。

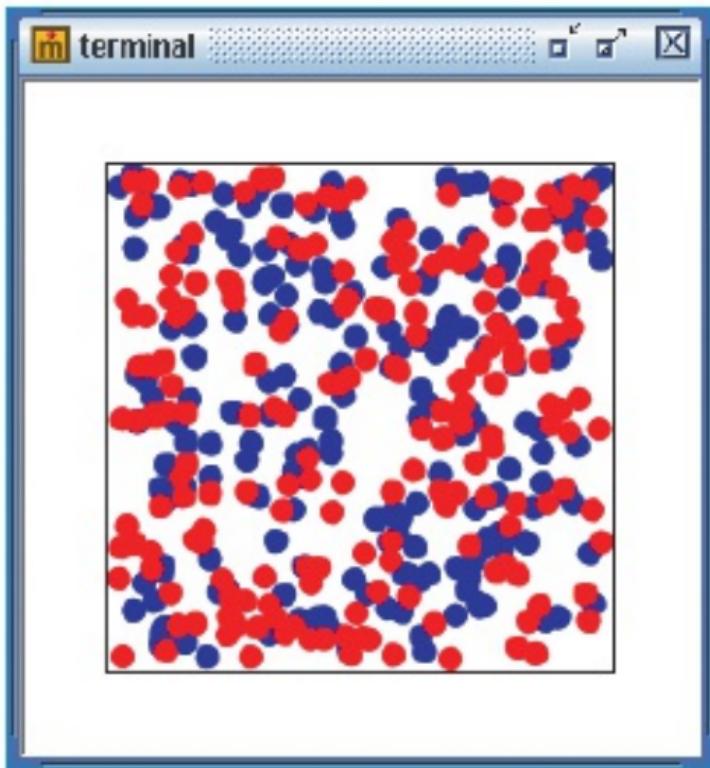


図11.4

マップ上には2種類のエージェントがばらついて存在しているでしょうか。何も見えないかも知れません。その場合は、停止ボタンを押してから、コントロールパネルのスライドバーを適当に動かし、再度、実行ボタンを押してみて下さい。

11.5 ルールを書き込む(その2)

続いて、エージェントのルールを書き込みましょう。ここでは、東行きエージェントについて詳しく説明しま

す。

なお、以下では、適宜、コメントをルールに付記することにします。ルールが複雑な場合、コメントを付記してあるとルール全体がわかりやすくなります（コラム「[ルールを見やすく](#)」を参照）。コメントの書き方は何種類かありますが、ここでは、「//」（半角スラッシュ2つ）を用いて、この記号より後の部分（その行のみ）にコメントを書き込むことにしましょう。なお、artisocはコメントを無視します。必ずしも以下の例と同じに書く必要はありません。

1. 東を向かせる

```
Agt_Init{
    My.Direction = 0 //東を向く
}
```

2. 正面・右・左の前方の対向エージェントの状況を調べさせます。ここでは、正面と左右45度の前方距離1の周囲を調べることにしましょう。全体ではやや長いルールですが、短い基本的ルールの繰り返しです。同じルールを繰り返すときの効率の良いルールの書き方は第26章で学びます。長くなりますが、ルールの入力の練習のつもりで、がんばってください。

正面前方、右前方、左前方の対向客の数をCNo、RNo、LNoという一時的変数で表すことにしましょう。

```
Agt_Step{
    Dim crowd As Agtset
    Dim CNo As Integer //正面前方の対向客数
    Dim RNo As Integer //右前方の対向客数
    Dim LNo As Integer //左前方の対向客数
    //正面前方を調べる
    Forward(1)
    MakeOneAgtsetAroundOwn(crowd, 1, Universe.terminal.westward,
```

```
False)

CNo = CountAgtset(crowd)

Forward(-1)

//右前方を調べる

Turn(-45)

Forward(1)

MakeOneAgtsetAroundOwn(crowd, 1, Universe.terminal.westward,
False)

RNo = CountAgtset(crowd)

Forward(-1)

Turn(45)

//左前方を調べる

Turn(45)

Forward(1)

MakeOneAgtsetAroundOwn(crowd, 1, Universe.terminal.westward,
False)

LNo = CountAgtset(crowd)

Forward(-1)

Turn(-45)
```

3. 状況に応じて前方に進み、そしてまた東を向くようにします。フローチャートに忠実に従って、ルールを書き込みます。

```
If CNo == 0 Then //正面前方にだれもいなければ
    Forward(1.2)
    My.advance = 1.2
Elseif RNo == 0 Then //右前方にだれもいなくて
    If LNo == 0 Then //左前方にもいなければ
```

```
If Rnd() < 0.5 Then
    Turn(-30)
Else
    Turn(30)
End if
Else //左にはいれば
    Turn(-30)
End if
Forward(1.2)
My.advance = 1
Elseif LNo == 0 Then //正面や右にはいるが左にはいなければ
    Turn(30)
    Forward(1.2)
    My.advance = 1
//前方にはどの方向にも対向客がいる場合
Elseif RNo > CNo And LNo > CNo Then //正面が一番すいていれば
    Forward(0.8)
    My.advance = 0.8
Elseif RNo > LNo Then //左の方がすいていれば
    Turn(30)
    Forward(0.8)
    My.advance = 0.7
Elseif RNo < LNo Then //右の方がすいていれば
    Turn(-30)
    Forward(0.8)
    My.advance = 0.7
Else //どちらも混んでいれば
    Turn(Rnd() * 30 - 15)
```

```
Forward(0.5)
My.advance = 0.5
End if
My.Direction = 0
Universe.advance = Universe.advance + My.advance
}
```

上のように、イフ文の「入れ子構造」を繰り返せば、かなり複雑な場合分けを表現することが可能になります。なお、入れ子構造を複雑にすると、場合分けを完了させる「End if」を書き忘れることがあります。注意しましょう。

なお、ここで、移動距離(つまりForward(D)のD)よりも駆方向速度(つまりMy.advance = V)が、左右前方に移動しながら進むときには直進の場合と比べて小さいことに注意して下さい。たとえば、斜め30度前方に1.2移動すると、正面方向には1移動しているという設定です($1.2 \times \cos 30^\circ$ はだいたい1.0)。ただしこれは厳密なものではありません。

また、最後の

```
Universe.advance = Universe.advance + My.advance
```

は、厳密に言うとエージェントの行動ルールではなく、シミュレーションの過程で全体状況を調べるための作業(具体的には、移動速度の集計)です。エージェントがモデル作成者に協力してくれている、と思って下さい。

4. 西行きエージェントのルールも忘れずに

東行きエージェントだけでなく、西行きエージェントについてもルールを書き込みましょう。この際、「コピー・アンド・ペースト」が便利です。その後、必要な修正をしましょう。西行きエージェントについては、西に向かうので、My.Direction = 180とする必要があります。また、対向する乗り換え客を調べるのですから、MakeOneAgtsetAroundOwn()で調べるエージェントをeastwardにすることも忘れないで下さい。

ではrush-hour(B)という名前を付けて保存して、実行してみましょう。

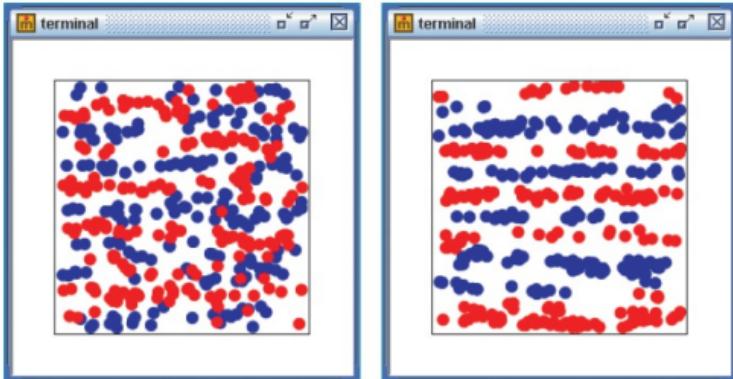


図11.5

新しく学んだ事項

- イフ文を複雑に組み合わせる
- 周囲の様子を調べる技巧的な方法
- コメントを付してルールを読みとりやすくする

練習問題

イフ文の使い方に慣れましょう。

11.1

11.5の3のエージェントのルール表記で登場したイフ文で、下の3重の入れ子構造の部分を2重に単純化してみる。

```
Elseif RNo == 0 Then
  If LNo == 0 Then
    If Rnd() < 0.5 Then
      Turn(-30)
    Else
      Turn(30)
    End if
  Else
    Turn(-30)
  End if
```

以下省略



複数の条件をひとつの条件式にまとめます。

11.2

この章で作ったモデルは、対向客を避けるルールが基本だった。今後は逆に同方向の客に追随するルールに変えてみよう。

乗換客は、自分と同方向に向う客が多い方向に移動しながら進む

果たして、スムーズな人の流れは現れるだろうか。

ルールを見やすくミスを少なくするためにも

ルールが複雑になると、何行もルールを書かなくてはなりません。ルールは後に見直すことを想定して、なるべく見やすくするように心がけましょう。ここでは、ルールエディタにルールを書き込む際の助言をいくつか挙げておきます。

◎ コメント

ルールエディタの中にコメントを書き込むと、artisocはその部分を無視してくれますので、自分なりの解説や一時的に実行したくないルールはコメントしておけば便利です。具体的には、'（アポストロフィ）か、//（スラッシュを2つ）を記述すれば、そこからその行の終わりまでがすべてコメントされて、ルールには反映されません。ちなみにコメントされたことがわかるように、色が緑へと変色します。

たとえば、

```
// Dim kazu as Integer  
Dim kazu as Double 'kazuは実数型で定義
```

と書けば、上の行全体と下の行の「kazuは実数型で定義」という記述は無視されます。

また、1行ではなく、数行にわたってコメント使用したければ、/* ~ */と記述します。こうすると、/*から */の間のルールをすべてartisocは無視してくれます。

上の例にあるように、ルールをコメント化すると無視されるので、ルールが正しいかどうかチェックするときに、いちいち削除せずに、コメントとすると便利です。

◎ 字下げ

設定 > 他の設定 の、ルールエディタの設定で「自動インデント」にチェックをしておけば（デフォ

ルトはそうなっています)、タブを使うと自動的に字下げ(インデント)するようになります。すべてのルールを左詰めで書いてしまうと見にくくなってしまうため、イフ文やフォ文を記述するときなどは、インデントを使用することをお勧めします。

◎ スペース

字下げと同様、適宜スペースを入れれば、ルールが見やすくなりますが、その際にはいくつか注意すべき点があります。ときどき見かけるミスは、特に次の3点を守らないときに起こっています。気をつけましょう。

- 変数や関数は分かち書きしてはいけない

例えば、Elseifを、間にスペースを入れてElseとifとに分けてしまうと、artisocはそれぞれ別なものと認識してしまいます。

- 関数名と括弧の間にはスペースは入れてはいけない。

たとえば、my.Direction = rnd ()*360というように、rndと()の間にスペースを入れると、エラーになります。関数名と括弧の間にはスペースを入れないように心がけましょう。ただし、括弧の中ではスペースを入れてもかまいません。たとえば、MakeAllAgtSetAroundOwn(my.Neighbor, 2, False)と記述してもエラーにはなりません。

- 変数と変数の間にはスペースを入れなければならない

aという変数と、bという変数を定義したとします。その後abとスペースを入れずに書けば、artisocは「未定義です」というエラー・メッセージを返してきます。つまり、aとbにスペースを入れないと、それぞれ独立した変数として認識されないので。逆に言えば、スペースが入った変数(たとえば「a b」)を定義することはできません。

(保)

第12章

エージェントの属性を豊富にする

12.0 「量より質」の豊かさを求めて

- エージェントの空間座標値(X, Y)に別の属性を与えられます
- artisocの空間をグラフとして利用しましょう
- 個々のエージェントの属性として「色」も追加できます
- エージェントの「色」は出力マップに表せます
- シミュレーション実行中に属性(色)が変わってもマップに現れます
- エージェントに色を付ける方法を学びましょう

12.1 属性の「一目瞭然」化

この章の目的は、エージェントの属性を豊富にする技法を学ぶことです。個々のエージェントに与えられる変数はそのエージェントの属性と見なすことができます。その意味では、エージェントにたくさん変数を追加することにより、いくらでも属性を豊富にできます。

artisocでは、空間の下にエージェント種を作ると、自動的に必ず、ID, X, Y, Layer, Directionという5つの変数が作されました。IDはまさに個々のエージェントにユニークな番号になりますから属性ですが、その他の変数も空間に関連したエージェントの属性として捉えることが可能です(ちなみに、Universeの直下に(空間を作らずに)エージェント種を作ると、IDのみ自動的に作られます)。実際、第1部のモデルでは、空間はoozoraとかhirobaとかgekijoとか何らかの意味で物理的な空間をイメージしてきました。そして、X, Yを空間の座標値としてルールの中で用いてきました。

しかしXとYをエージェントの空間とは関係ない2つの属性であると見なすことも可能です。そのように見なすと、マップ出力も地図(2次元空間の表示)ではなく、横軸X・縦軸Yのグラフと見なすことになります。

つまり、エージェントが空間の下に存在しているからと言って、空間上を動き回る存在としてエージェントを捉える必要はないのです。エージェントは物理的空間に張り付いているものだという固定観念から自由になるだけで、エージェントの属性は豊富になります。

もっとも、見方の質的転換があればエージェントの属性が豊富になるということを指摘することだけが、この章の目的ではありません。もちろん、「目から鱗が落ちる」ことは重要ですが、属性の表し方はまだ他にもあります。実際、この章のハイライトは、エージェントの属性として色を指定する手法を学ぶことです。出力マップの上で個々のエージェントに色を自由に付けることができるだけでなく、シミュレーションの過程で属性=色を変えることも可能です。いろいろな活用法があるので、「見かけ」の問題だとばかりにしないで、しっかりとこの手法を身につけて下さい。

12.2 人工国家の人口と経済

人工社会のエージェントは物理的空間を動き回るものだと思いこむ必要はありません。このような固定観念を早く打ち壊すために、この段階で、artisocの中でUniverseの下に作る空間を、抽象的なものとして捉えることに慣れておきましょう。

世界の全100カ国は、初期状態では、全て、人口も1単位、経済規模も1単位だが、毎ステップ、人口は2%前後で増加し、経済規模は2%前後で成長する。

わずかの違いのせいで、どれだけ国家間格差が広がるのでしょうか。もちろんエージェントは国ですが、自動的に作られる変数のうち、Xで人口を、Yで経済規模を表すことにします。これにより、出力マップは、100カ国の人団と経済規模を示すグラフになります。

そこで、graph(「ループしない」に設定)という空間にnationエージェントを(エージェント数は100)作り、人口増加率(PGR)と経済成長率(EGR)の2つの変数(実数型)を追加しておきます。また、マップ出力の設定もしておきます。このモデルは、national-sizeという名前で保存することにしましょう。

ところで、ここでgraphという名前を付けた空間には、worldという名前を付けてもよさそうですが、エラー

になります。というのは、worldとかuniverseとかは、artisocで特定の用途に使うために予め指定される「予約語」で、モデルを作る私たちが勝手に使えない単語です。他にも、IfやForなども、変数名などで勝手に使えない単語です（コラム「[予約語とその勝手な使用の禁止](#)」を参照）。

Universeのルールには何も書かず、nationのルールエディタは次のようにしてみます。

```
Agt_Init{
My.X = 1
My.Y = 1
My.PGR = 0.02 + (Rnd() - 0.5) / 50
My.EGR = 0.02 + (Rnd() - 0.5) / 50
}
Agt_Step{
My.X = My.X * (My.PGR + 1)
My.Y = My.Y * (My.EGR + 1)
If My.X >= 50 Or My.Y >= 50 Then
    ExitSimulationMsgLn("Completed after " & GetCountStep() & " steps")
End if
}
```

人口（経済）の成長は、前の期の値に増加率（成長率）をかけることで、当期の値を求めることができます。新しい文法は何も使っていません。My.QQQ = My.QQQ + Rという形の代入文は、既に何回も使ってモデルを作りましたが、足し算だけでなくかけ算でも、左辺と同じ変数を右辺に用いる代入文が使えます。このモデルのようにエージェント間の相互作用がなければ、各ステップでルールを実行しようとするとき、My.X（あるいはY）には1ステップ前の値が入っていますから、My.X(Y) * (My.PGR(EGR)+1)でそのステップにおける新しい人口（経済規模）になります。それをMy.X(Y)に代入すれば良いわけです。

最後のイフ文はおなじみの終了のさせ方ですね。どれか一つの国の人団体が経済規模が50単位になったところでシミュレーションは終わりです。条件式に「Or」が用いられているので、少し複雑な形になっています。

それでは、上書き保存をしてから、実行してみましょう（[図12.1](#)）。

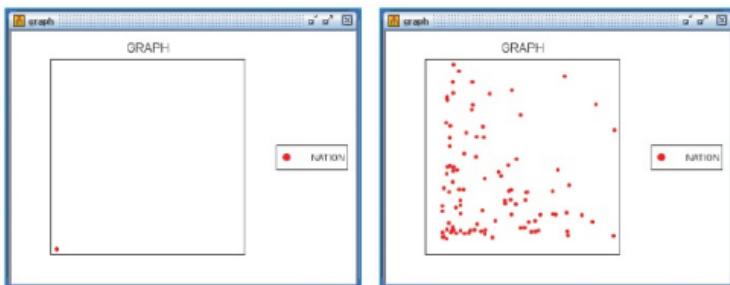


図12.1

実行しても最初のうちは、ほとんど変化が見られないのに、やがて急速に増加していく様子が見られたと思います(図12.1右)。このモデルでは、確率的に、最も大きく増加しても、毎ステップ0.03(つまり3%)ですから、おおよそ133ステップか134ステップで、グラフ上限の50に到達するはずです。終了したときのメッセージを読んで、だいたいそうなっていることを確認して下さい。

このモデルは、人口や経済規模が幾何級数的に増加していくことを実感するためだけの単純なものです。わたしたちが言うところの「空間」とは、必ずしも物理的な空間を意味しない、という点が重要です。空間の出力マップは、リアルタイムの人口・経済規模の分布図です。そして、X軸が人口、Y軸が経済規模を表していますから、経済水準(経済規模／人口)は45度の傾きの直線が同一水準を表し、左上にシフトするほど高水準、右下にシフトするほど低水準です。

言い換えると、XとYという変数は、上手に工夫すると、エージェントの属性に関わる相対的関係をマップ出力という出力設定機能を利用して、リアルタイムでビジュアル化することができるのです。

12.3 経済水準を色で表す

いよいよ、エージェントの属性に「色」を持たせる方法を学びます。属性のビジュアル化(マップ出力されたエージェントの色を属性に応じて変える)に不可欠な技法です。

上のモデルでは、経済水準は経済規模と人口から計算することができますが、これも国家の属性とみ

なすことができます。そこで、経済水準をエージェント自身に計算させて、水準をグラフ上で色によって表示することにしましょう。エージェントを特定の色でマップ上に表すには、エージェントに特定の色を属性として与える変数を追加することと、その変数値をマップ上に出力するための設定が必要です。

では、その作業に取りかかりましょう。nationエージェントにecolevelという変数を「整数型」(実数型ではありません!)で追加して下さい。artisocでは、「色」を値に持つ変数は整数型と決められています。普通の意味の経済水準は経済規模／人口ですが、ここではそのような数値ではなく、「色」で経済水準を表すことにして、ecolevelは整数型にします。

次に、出力設定で、マップ出力を選択します。ただし、既にgraphというマップの出力を設定してありますから、その編集をクリックして下さい。そして、マップ要素リストにあるnationの編集をクリックして下さい。すでにおなじみの画面が開きます。

ここで、エージェント表示色をデフォルトの「固定色」ではなく、「変数指定」にしてください。すると変数の選択肢が出てきますから、ecolevelを選択します([図12.2参照](#))。

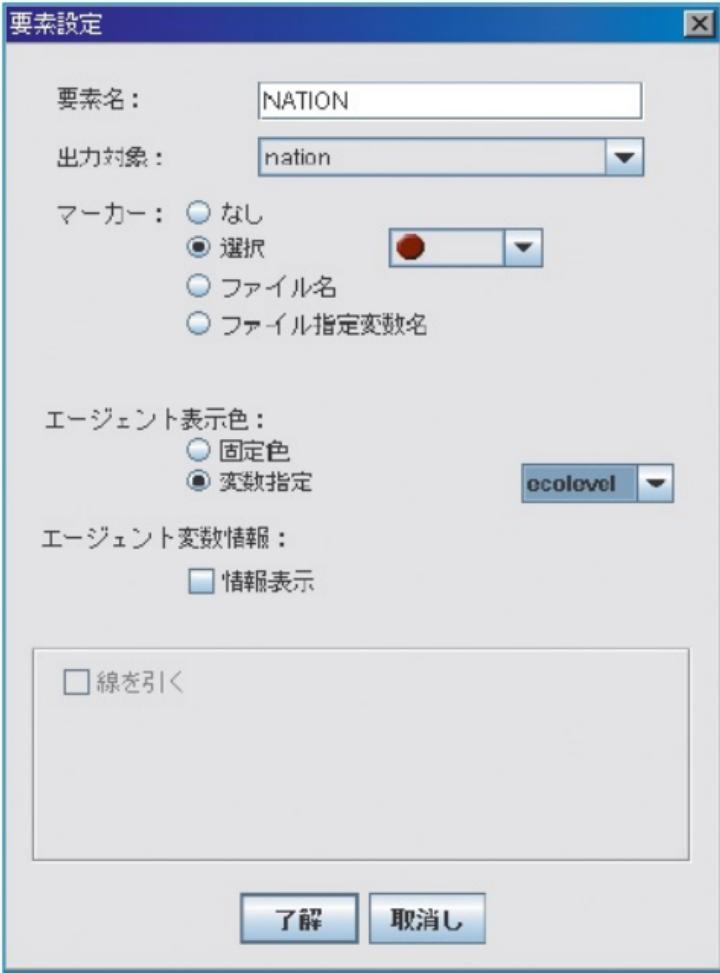


図12.2

これからエージェントのルールエディタで、ecolevelに色の属性を与える方法を学びます。ここでは作業を簡単にするために、経済水準を、次のように設定します。

経済規模が人口より5単位以上大きくなった国の色を赤に、逆に人口が経済規模より5単位以上大きくなった国の色を青にする。どちらにもあてはまらない場合は緑にする。

このルールは、次のようにになります。エージェントルールエディタのなかのAgt_Step{ }のセクションで My.X, My.Yを計算させている箇所の次に続けて、書き込んで下さい。

```
My.X = My.X * (My.PGR+1)
My.Y = My.Y * (My.EGR+1) // 既にここまで書き込み済み
If My.Y >= My.X + 5 Then
    My.ecolevel = Color_Red
Elseif My.X >= My.Y + 5 Then
    My.ecolevel = Color_Blue
Else
    My.ecolevel = Color_Green
End if
If My.X >= 50 Or My.Y >= 50 Then //ここから下は書き込み済み
    ExitSimulationMsgLn("Completed after" & GetCountStep() &
"steps")
End if
```

では、このように修正したモデルを、national-size(B)という名前で保存してから新しいルールを実行して、国の色が変わる様子を観察して下さい(図12.3)。

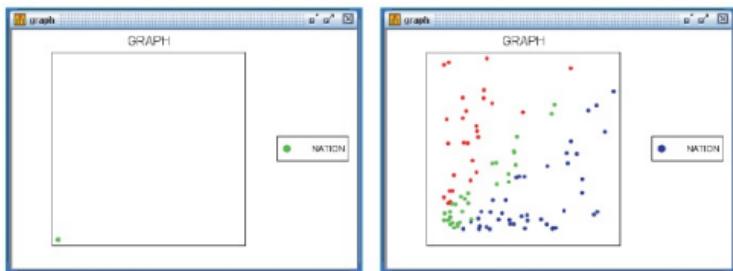


図12.3

artisocでは、属性に色を与える場合、以上の3色を含めて、Color_XXXという形式で以下のようない8色を指定できます。文字として書かれていますが、実は整数值です。

Color_Red 赤, Color_Green 緑, Color_Blue 青, Color_Yellow 黄, Color_Cyan 水色, Color_Magenta 紫, Color_Black 黒, Color_White 白

「色」を属性として与えると、シミュレーション実行中に、エージェントの状態(変数、属性)が変わるようなとき、それをビジュアル化する際にとても便利です。もっとも、上の例では、モデルの前提から、1回色が変わるともう変化しません。その意味では、あまりスマートな利用法ではありません。

12.4 もっと微妙な色で属性を表す

artisocでは、微妙に色を与えることもできます。それは、「RGB関数」と呼ばれる、色の指定方法をルールのなかで用いることによって行なうのです。RGBとは、光の三原色である、赤(Red)緑(Green)青(Blue)のこと、この3種類の光源をミックスすることにより、さまざまな色を作り出せることに由来しています。RGB関数はRGB(xxx,yyy,zzz)のような形をしています。ここで、xxx,yyy,zzzは、いずれも0以上255以下の整数です。そして

白 – RGB(255, 255, 255)

赤 – RGB(255, 0, 0)

緑 – RGB(0, 255, 0)

青 – RGB(0, 0, 255)

黒 – RGB(0, 0, 0)

のように色と関数とが対応しています。そしてRGB関数自体も整数の値をとります。そのため、色を表す変数は、整数型にする必要があるのです。詳しくは、コラム[「色の秘密」](#)を参照して下さい。

それでは、RGB関数を利用して、人口増加と経済成長に応じて色がなめらかに変化するようにしてみましょう。

初めは黒っぽいが、経済成長が相対的に高ければ赤っぽく、人口増加が相対的に高ければ青っ

ぼくする。

12.3で書き込んだ、特定の色をmy.ecolevelに代入するイフ文の代わりに

```
My.ecolevel = RGB(5 * Int(My.Y), 0, 5 * Int(My.X))
```

という代入文を1行記述するだけで、おおよそ、このルールを再現できます。

つまり初期状態は、RGB(5, 0, 5)でほとんど黒です。人口増がなく経済だけ成長するとRGB(250, 0, 5)となり、ほとんど赤です。経済成長がゼロで人口が増えるだけだとRGB(5, 0, 250)となり、ほとんど青ですね。

それでは、イフ文全体を削除して、代わりにRGB関数の代入文を書き込みましょう。修正したモデルは、national-size (C)という名前で保存して、実行しましょう([図12.4](#))。

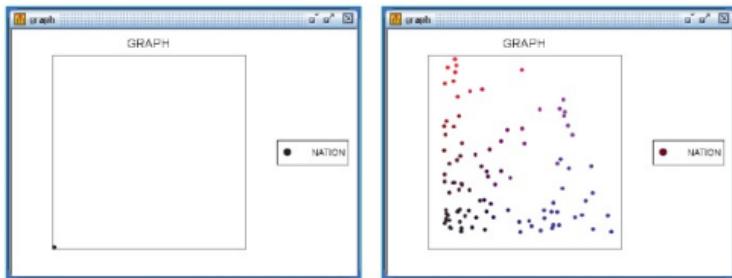


図12.4

もう少し、技巧的な「色付け」手法を紹介しておきます。

経済水準が1(My.XとMy.Yとが同単位)のときは緑、そして経済水準が高くなるにつれて(左上隅に近づくほど)赤っぽく、逆に低くなるにつれて(右下隅に近づくほど)青っぽくなるように表示する。

いろいろなルールの書き方があり得ますが、ここでは下記のように、 $(My.Y - My.X) / (My.X + My.Y)$ というプラス1からマイナス1の範囲の値をとる指標(el)で経済水準を表します。このように設定すると、経済水準は、高ければプラス1に低ければマイナス1に近づきます。このように指標化するのは、通常の $My.Y / My.X$ で指標化すると数十から数十分の一の間の値をとるので、色と対応づけようとするときに扱いにくいからです。なお、分子の正負に対応した「場合分け」に注意して下さい。

```
Dim el As Double  
el = (My.Y - My.X) / (My.X + My.Y)  
If My.X <= My.Y Then  
    My.ecolevel = RGB(Int(265 * el), 255 - Int(265 * el), 0)  
Else  
    My.ecoLevel = RGB(0, 255 - Int(265 * (-el)), Int(265 * (-el)))  
End if
```

人口増がゼロで最速経済成長だと、左上隅(1, 50)に到達し、そこはelが約0.96ですからInt(265*el)は255になります。したがってRGB(255, 0, 0)に対応しています。逆の場合だと、右下隅(50, 1)になり、RGB(0, 0, 255)になります(初期値が(1, 1)ですから、(0, 50)や(50, 0)には到達しません)。

この新しいルールに変えたモデルは、national-size (D)という名前で保存してから、実行してください。国の色がなだらかに変わっていく様子を観察して下さい([図12.5](#))。



図12.5

[12.5 エージェント種が異なるのか、属性が異なるのか](#)

マルチエージェント・シミュレーションのモデル化に際して、行動様式の異なるエージェントは別種のエージェントとして設定してきました。たとえば、東に向かう乗降客と西に向かう乗降客、1セント硬貨(ペニー)と10セント硬貨(ダイム)、人間(飼い主)とペット、などです。そうすれば、出力マップで異なる色で表示てきて、識別が容易になります。

しかし、属性の違いを色の違いで表す技法を学んだ私たちにとって、エージェントの種類の違いを属性の違いと見なすことも可能です。つまり、乗降客(東向きか西向きか)、コイン(1セント硬貨か10セント硬貨か)、広場を動き回る生き物(人間かペットか)というふうに、属性の異なる一種類のエージェントと考えることです。属性の違いを出力マップで表せるのはもちろん、属性の違いに応じて行動を変えさせることもできます。

種が異なるのか属性が異なるのかという問題は、どちらが正しくどちらが間違っているのか、というものではなく、モデルを作る際の柔軟性や行動ルールの類似性などにしたがって、適宜判断すべきことです。もちろん、別種のエージェントとしてルールを書く場合と、同一種エージェントとしてルールを書く場合とは、同じモデルであっても、ルールの書き方は異なってきます。

新しく学んだ事項

- マップを2次元グラフとして利用する
- 定数(整数型)としての色の指定(Color_Redなど)とマップ出力
- 実行中にエージェントの色属性を変える
- RGB関数
- 勝手に使えない単語(予約語、禁止語)の存在

練習問題

色の扱い方に慣れましょう。

1×1 の大きさの空間にエージェントをひとつだけ作り、コントロールパネルを使って、そのエージェントの色をマップ上で自由に変えてみよう。



Universeに3種類の整数型変数を作る。

12.2

national-size(C)モデルは、経済水準を $My.ecolevel = \text{RGB}(5 * \text{Int}(My.Y), 0, 5 * \text{Int}(My.X))$ により色づけしているので、初めは黒に近い色である。初め(左下)は赤で、経済が成長するにつれて(左上に向かうほど)紫色に、人口が増加するにつれて(右下に向かうほど)黄色に変化するようにRGB関数を書いてみよう。なお、紫は $\text{RGB}(255, 0, 255)$ 、黄は $\text{RGB}(255, 255, 0)$ 。ちなみに、このようにルールを書くと、経済も人口も増大する(右上に行くと)と何色になるだろうか。

予約語とその勝手な使用の禁止

予約語とは、artisoc側であらかじめ定義されているために、ユーザが変数名や関数として自由に使えるない単語のことを意味します。たとえば、"if" という単語をエージェントとして定義してしまうと、条件分岐の際に使う"if" と区別ができなくなってしまって、artisocが混乱してしまいます。このようなことを避けるために、予約語が決められているのです。また、組み込み関数もユーザが自由に使用することはできません。

試しに、新しい空間を作成する際に空間名に「Space」と入力してみて下さい。左下に“名前に予約語が指定されています”と表示されて、入力できないと思います。また、ルールの中に一時変数として予約語を定義してみると、モデルは実行されず、左下に“……は予約語なので変数名として利用できません”と表示されます。

このように、ユーザ側が予約語を使用しようとするとartisocは警告を鳴らしてくれるのです。artisocの予約語は60以上に及ぶため、全ての予約語はここでは取り上げませんが、代表的なものを以下に挙げておきます。全ての予約語を知りたい人はマニュアルを参照して下さい。

(保)

予約語の代表例

定数	True, False, COLOR_RED, COLOR_BLACKなど
型名	Integer, Double, AgtType, Agent, Agt, Spaceなど
演算子	And, Or, Modなど
実行制御	If, Else, End, Step, Next, While, For, Eachなど
識別子の宣言や参照	Dim, As, Function, Universe, My, Agt_Initなど

第13章

周囲のエージェントから影響を受ける

13.0 周囲の影響から逃れられません

- 病気の流行をモデル化しましょう
- 周囲のエージェントの状態が重要です
- 他のエージェントの属性を調べさせます
- エージェントを取り出して調べる「フォ・イーチ文」を学びます

13.1 周囲のエージェントを詳しく調べる

周囲の状況を認識(観察)して、自分自身の行動を変化させる技法の基本は、既に第6章で学びました。それは、周囲に他のエージェントがどれだけ存在しているかによって自分の行動を選ぶ、というものでした。この章と次章で、その技法を一般化します。具体的には、周囲にいる他のエージェントの属性の状態に依存して、自分自身の行動や属性に変化が生じる、という技法です。

他のエージェントから影響を受ける、ということは、モデルを作る立場から見ると

1. 周囲を認識させて、周囲にいる他のエージェント(複数でも)をリストアップする。
2. リストにあるエージェントたちの属性を認識する。
3. 認識した属性に応じて、自分の行動や属性を変化させる。

という3段階の作業をエージェントにさせる必要があります。1については、既に第6章で学んでいます。また、3は既に何回も登場した「場合分け」の応用です。したがって、2が新しい問題です。artisocでは、周囲のエージェントの属性を調べることができる機能があります。その技法をここで学びます。

13.2 病気の流行をモデル化する

周囲から影響を受ける現象で昔から注目されていたのが病気の流行です。確率過程の数理モデルもたくさんあります。近年流行のネットワーク理論でも研究されています。ここでは人工社会の観点から、周囲に病人がいると病気になるモデルを考えます。まずは、感染症が流行するモデルの最も基本的な形から始めましょう。

社会に風邪をひいている人(患者、赤色のエージェント)が少数いる。健康な人(水色のエージェント)は、自分の周囲に風邪をひいている患者が多いと、自分も風邪をひく(赤くなる)。

この場合、赤色エージェントと水色エージェントという2種類のエージェントがいるわけではありません。單一エージェント種に、風邪をひいている(「赤色」の属性)か健康(「水色」の属性)かを区別する属性変数を持たせます。

まず、いつものように準備作業から始めましょう。なお、エージェント数をコントロールパネルで増減できるようにします。

1. Universeの下にsocietyという空間(デフォルト値を利用)とpopという人口数を表す整数型変数を作る
2. 空間の下にpersonをエージェント数0(デフォルト値)で作る。personには健康状態を表すconditionという整数型変数を追加する
3. 出力設定(マップ出力)する。マップ要素を追加する際、色をconditionによる「変数指定」にする
4. コントロールパネル設定する。対象はpopで、100から2000までの範囲を100の刻みでコントロールできるようにする
5. モデルをinfectionという名前をつけて保存する

最初にpersonエージェントをpopで指定された数だけ生成し、societyにばらまく作業をします。なお、最初に風邪をひいている人の割合を1パーセントとしましょう。

```

Univ Init{
Dim i As Integer
Dim people As Agtset
Dim one As Agt
For i = 0 To Universe.pop - 1
    one = CreateAgt(Universe.society.person)
    If Rnd() < 0.01 Then //1%の確率で風邪をひく
        one.condition = Color_Red
    Else
        one.condition = Color_Cyan
    End if
Next i
MakeAgtset(people, Universe.society.person)
RandomPutAgtset(people)
}

```

Universeのルールで、エージェントを生成して空間にばらまく技法は、第7章で既に学び、第11章でも使いましたが、ここでは1点だけ新しいルールを使いました。oneというエージェント型の一時的変数を利用するルールです。

ルールの冒頭に、一時的変数を使うための型宣言文Dim one As Agtがあります。oneという変数はエージェント型(As Agt)になります。エージェント型変数というのは、特定のエージェント1体だけを値としてとることのできる変数です。エージェント集合型変数は、エージェントを何体でも値としてとることができるので、エージェント型変数とエージェント集合型変数とを混同しないようにして下さい。

CreateAgt()が実行されて生成されたエージェントを、ただちにoneに代入します。こうすることにより、生成されたばかりのエージェントに対し、1%の確率で風邪をひいている状態に設定する事が可能になります。one.conditionという表現が出てきます。これは初めての表現方法です。これは、ここだけで使われているoneという仮の名前のエージェント(その実体は、personというエージェント種のエージェント)が持っているconditionという変数を意味します。

上書き保存してから実行ボタンを押してみて下さい。うまく初期配置されているでしょうか([図13.1](#))。

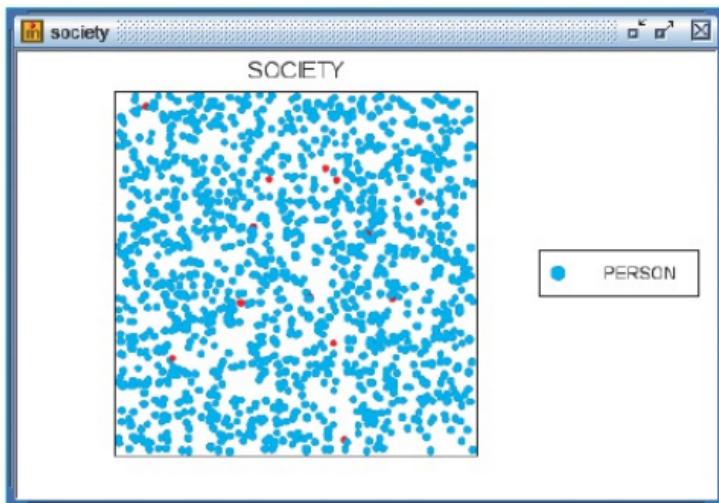


図13.1

13.3 「フォ・イーチ文」の登場

personエージェントのルールは図13.2のフローチャートのようになります。周囲にいるエージェントを調べる点に注意して下さい。

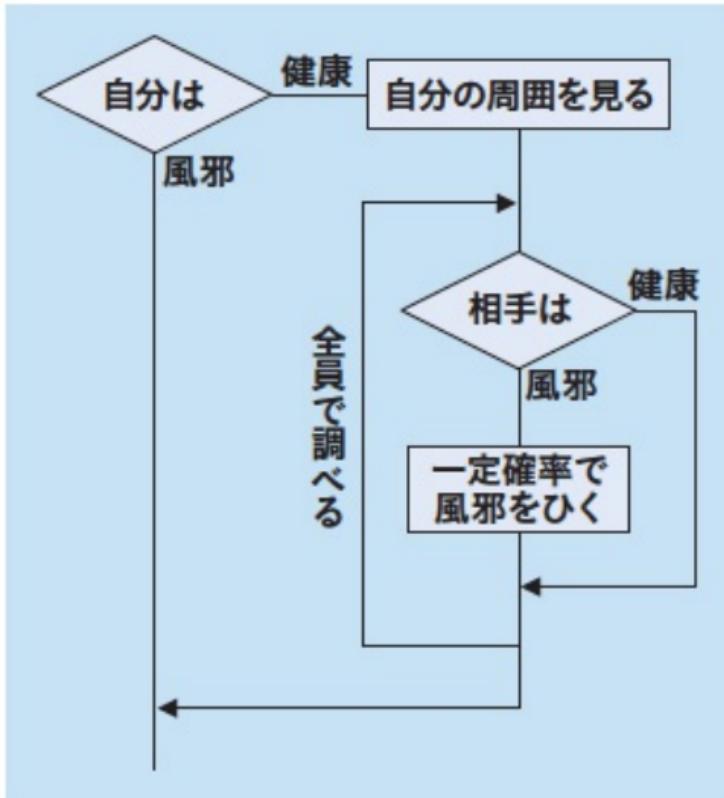


図13.2

周囲に風邪をひいている人が1人いる場合、3割の確率で自分に風邪がうつるものとしましょう。新しいルール表記が登場します。

```

Agt_Step{
Dim neighbor As Agtset
Dim one As Agt
If My.condition == Color_Red Then //風邪をひいているなら
//何もない
Else //風邪をひいていないなら

```

```
MakeAllAgtsetAroundOwn(neighbor, 2, False)
For each one in neighbor //neighborの中から順次取り出す
  If one.condition == Color_Red Then //風邪ひきなら
    If Rnd() < 0.3 Then
      My.condition = Color_Red
    End if
  End if
  Next one
End if
}
```

ここで風邪をひいていない場合に実行するルール(ElseとEnd ifとの間)に

```
For each one in neighbor
  QQQQQ
Next one
```

というルール表記があります。初めて登場したルールです。これは、neighborというエージェント集合型変数にしまわれているエージェントをoneという仮の名称(変数型はエージェント)で1つずつ順番に全て取り出してきて、個々のoneについて、QQQQQというルールを実行しなさい、というルールです。

このルール表記は、

```
For i = m To n
  QQQQQ
Next i
```

という表記(フォ文)に似ていますね。そこで

```
For each A in S
  QQQQQ
Next A
```

(Aはエージェント型変数、Sはエージェント集合型変数)という形の新しい構文を「フォ・イーチ文」と呼びます。

neighborの中にはMakeAllAgtsetAroundOwn(neighbor, 2, False)という操作によって、視野2の範囲にある自分以外のエージェントが全て含まれています(既に第6章で学びました)。それらをoneという仮の

名称で1つずつ取り出してきて、QQQQQというルールを当てはめるのです。全てのエージェントを取り出し尽くしたら、この作業は完了です。

なお、モデルに複数のエージェント種が存在していて、1つのエージェント種のみに注目するときには、`MakeOneAgtsetAroundOwn()`(第6章で学びました)を使います。

QQQQQの箇所はイフ文になっています。周囲(視野2の広さ)にいる人たちが病気かどうか調べて、病気ならば0.3の確率で自分も病気になる、というルールです。ただ、若干注意を要する点は、周囲に患者が1人だけだと風邪をひく確率は30%ですが、2人だと約50%，3人だと約65%…と増えていきます。

では、`infection(B)`という名前を付けて保存してから実行してみましょう。

人口をコントロールパネルでさまざまな値に設定して下さい。人口の多寡に応じて(つまり社会の人口密度の大小に応じて)、風邪の流行り方の違いが生じることが分かると思います([図13.3](#)参照)。ステップ実行ボタンを押して、1ステップずつ変化を眺めると流行していく様子(あるいは流行が止まる様子)がよくわかるかも知れません。

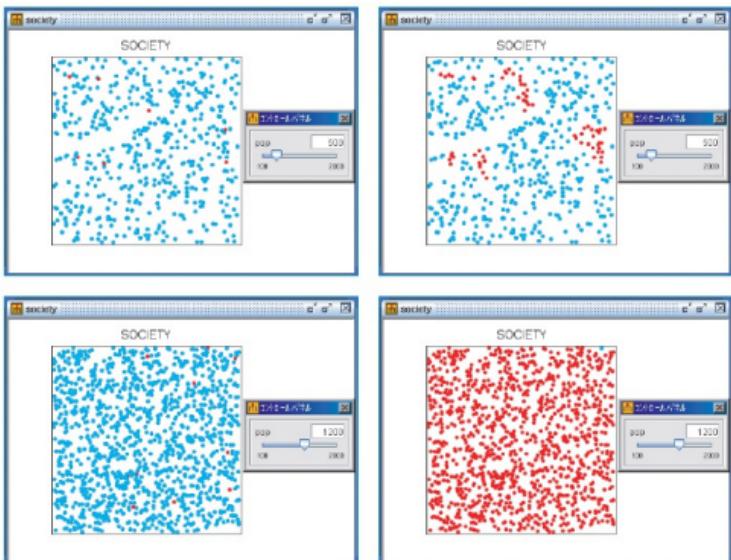


図13.3

13.4 病気の流行を把握する

風邪はどのように流行していくのでしょうか。第8章で学んだ時系列グラフの出力で、流行現象を見てみましょう。そこでは、各エージェントのルールで集計作業をさせていました。しかし、フォ・イーチ文を使うと、Universeのルールで全エージェントについて一括して集計が簡単にできます。ここでは、フォ・イーチ文の練習を兼ねて、以下のようなルールにしましょう。

1. Universeに患者数をカウントするnumber(整数型)を追加する。
2. 次のようなルールを書き込む。

```
Univ_Step_End{
Dim people As Agtset
```

```
Dim one As Agt
Universe.number = 0
MakeAgtsetSpace(people, Universe.society)
For each one in people
    If one.condition == Color_Red Then
        Universe.number = Universe.number + 1
    End if
Next one
}
```

3. 時系列グラフの出力設定をする。出力値としては、患者数そのものではなく、全人口に占める病人の割合を用いることにしましょう。

```
100 * Universe.number / Universe.pop
```

この時系列グラフの出力が加わったモデルは、infection(C)という名前を付けて保存してから、実行してみましょう。マップよりも全体の様子が把握できるのではないかでしょうか。時系列グラフの形が、いわゆるS字型カーブ(ロジスティック曲線)に近いことを確認しましょう。実は、上のルールでは、初期状態(ステップが0のとき)の感染率を集計していないので、立ち上がりが本当の状態よりきつい勾配になっています。

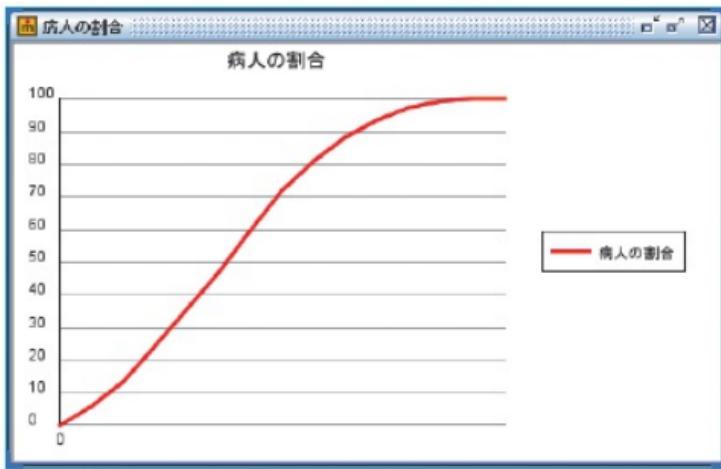


図13.4

13.5 現実の流行現象に近づける

上のモデルでは、一度風邪をひくとひきっぱなしになってしまいます。そこで、もう少し現実的な仮定を設けましょう。

風邪をひいても7ステップ経つと健康になる

(personにステップをカウントするための整数型変数cureを追加し、風邪をひいている場合に数が増えるルールにする)。

人々は動き回る(健康な人だけでなく、風邪をひいている人も)

(Direction, Turn(), Forward()の活用)。

初期状態の感染率を操作できるようにする

(Universeに実数型変数initratioを追加し、コントロールパネルを設定する(0と0.1の範囲を0.01間隔で))。

これら3点の改良は、第1部で学んだルールだけでできます。infection(C)モデルに変数やルールを追加して、infection(D)という名前を付けて保存しましょう。

初期設定のルールは、次のように変わります。薄字の部分は、既に書き込んだルールです。

```
Univ_Init{
(前略)
For i = 0 To Universe.pop - 1
  one = CreateAgt(Universe.society.person)
  one.Direction = Rnd() * 360
  one.cure = 0
  If Rnd() < Universe.initratio Then
    one.condition = Color_Red
  Else
    one.condition = Color_Cyan
  End if
Next i
MakeAgtset(people, Universe.society.person)
RandomPutAgtset(people)
}
```

エージェントのルールについては、追加箇所を指摘しておきましょう。

```
Agt_Step{
(前略)
If My.condition == Color_Red Then
  My.cure = My.cure + 1
Else
(中略)
End if
```

```
Turn(Rnd() * 20 - 10)
Forward(1)
If My.cure >= 7 Then //7ステップで全快
    My.condition = Color_Cyan
    My.cure = 0
End if
}
```

患者が回復するルールを追加すると、病気の流行現象はどのように変わってくるでしょうか。上書き保存してから、コントロールパネルをいろいろと操作しながら、実行して下さい。

新しく学んだ事項

- Universeのルールの中で、生成したエージェントの初期設定をする
- 周囲にいるエージェントを全て調べる
- フォ・イーチ文
- エージェント型変数とその使い方
- エージェント型変数をエージェントのように使う(たとえばone.condition)
- フォ・イーチ文を利用して、Universeのルールで一括集計する
- ステップをカウントして、何ステップおきにしか実行しないルールを書く

練習問題

infection(D)モデルのままだと、風邪が治っても、またすぐひくかも知れません。次のような追加的ルールも、たとえば「condition」変数がとり得る色を増やすことにより、今まで用いた文法で作れます。

13.1

いったん風邪が治れば(Color_Green)、しばらくは(たとえば14ステップ)ひかない。



新しい変数は不要です。cureを「使い回し」できます。

13.2

病気にかかっている間は一定の確率で死亡(Color_Blackに変わる)し、死亡したら動かない。



ルールは毎ステップ実行されます。7ステップ目に生き返ったりしないように。

第14章

特定のエージェントから影響を受ける

14.0 誰でもよいわけではありません

- 結婚相手をさがすチョウをモデル化します
- 個別の相手の属性が重要です
- エージェントの集合からエージェントを1体だけ選ぶ技法を学びます

14.1 エージェントをピックアップする

フォ・イーチ文を使うと、エージェント集合型変数の中にリストアップされた全エージェントを順番に1つずつ取り出してきて、その内部状態(属性)を調べ、その結果に応じて、自分の状態や行動を変えることができました。しかし場合によっては、全てのエージェントではなく、どれか1つのエージェントだけを取り出し、その属性を調べたい場合があります。

この章では、エージェント集合型変数の中のエージェントから1つだけを選び出す技法を使うモデルを作ります。周囲にいるエージェントをエージェント集合型変数の中にリストアップするという技法は前章と同じです。リストの中から1つを選び出す技法が、この章のポイントです。

14.2 結婚相手をさがすモデルを作る

周囲にいるエージェントから結婚相手をさがすことは、特定のエージェントを選び、その属性次第で自分の行動を変える典型例でしょう。異性かどうか、未婚かどうか、相性がよいか、などチェックします。ここでは本物の人間をモデル化するのではなく、もっと単純な行動をするチョウをモデル化しましょう。

チョウがキャベツ畑で交尾相手を求めて飛び交っている。メスはキャベツの葉にとまっている(簡略化)

のために）。近くにオスが来てとまるとき、つがいになる。オスはふらふらと飛び回っているが、近くにチョウを見つけると接近する。しかし、そのチョウがオスだったり、すでにつがっているメスだったりすると、他のチョウを求めて飛び去る。もし独身のメスの場合には、そのチョウの交尾相手としてそこにとどまる。

本物のチョウの行動はもう少し複雑ですが、この程度のそれなりに「リアル」っぽいモデルで十分でしょう（ここでの目的はチョウの行動様式を学ぶことではなく、人工社会構築の技法の習得ですから）。

では例によってモデル作りの準備作業から始めましょう。

1. Universeの下にcabbagefield空間をデフォルトで作り、その下にbutterflyエージェントをエージェント数60で作る。
2. butterflyには整数型変数conditionを作る。
3. cabbagefieldをマップ出力するように設定する。マップ要素にbutterflyを追加して、出力する色を変数指定(condition)に設定する。
4. モデルをmatingという名前で保存する。

14.3 ルールの大枠

ルールを書き込む作業に入りますが、このモデルではUniverseのルールエディタには何も書き込みません。

エージェントのルールは複雑ですが、要点は次のとおりです。

1. オスとメスの2種類のエージェントを作るのはなく、1種類のエージェントを、conditionという変数を利用して、「オスかメスか」や「未婚か既婚か」という属性を用いて区別する。独身オスを赤色(Color_Red)で表し、既婚オスを黄色(Color_Yellow)で表す。そして、独身メスを紫色(Color_Magenta)、既婚メスを青色(Color_Blue)で表す。
2. メスはオスかメスかに反応するが、オスは性別だけでなく、メスの状態(既婚か未婚か)にも反応する。

3. 独身メスをみつけたかどうかで、オスの行動パターンが変わる。
4. 自分の周囲にいるチョウを認識するために、一時的なエージェント集合型変数neighbor、その中の個々のチョウを選び出すために一時的なエージェント型変数oneを宣言しておく。

それでは、初期状態として、チョウをキャベツ畑にランダムに配置し、未婚オスと未婚メスとをランダムに半々の状態にしましょう。

```
Agt_Init{  
My.X = Rnd() * 50  
My.Y = Rnd() * 50  
If Rnd() < 0.5 Then  
    My.Direction = Rnd() * 360  
    My.condition = Color_Red  
Else  
    My.Condition = Color_Magenta  
End if  
}
```

上書き保存してから実行ボタンを押してみましょう。赤や紫のエージェントが散らばっているでしょうか（[図14.1](#)）。

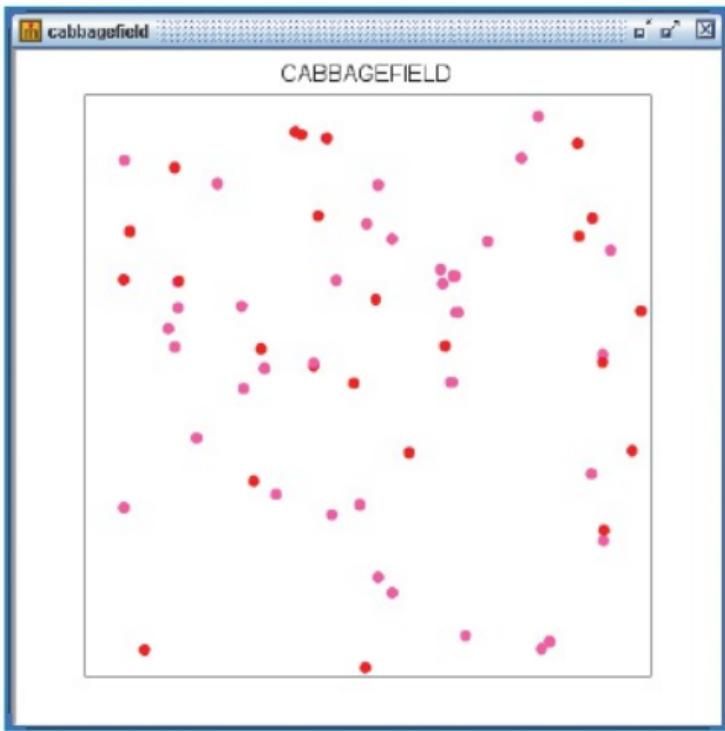


図14.1

以下では、エージェントの行動ルールの大枠を示して、メスとオスの行動ルールは個別に説明します。
大枠は次の通りです。

```

Agt_Step{
Dim neighbor As Agtset
Dim one As Agt
If My.condition == Color_Magenta Then //未婚メスなら
//（メスのルール、下記に詳述）
Elseif My.condition == Color_Red Then //未婚オスなら
//（オスのルール、下記に詳述）
End if

```

}

14.4 メスのルール

チョウエージェントは性別により、行動パターンが大きく異なります。まず、メスの行動ルールをまとめよう。

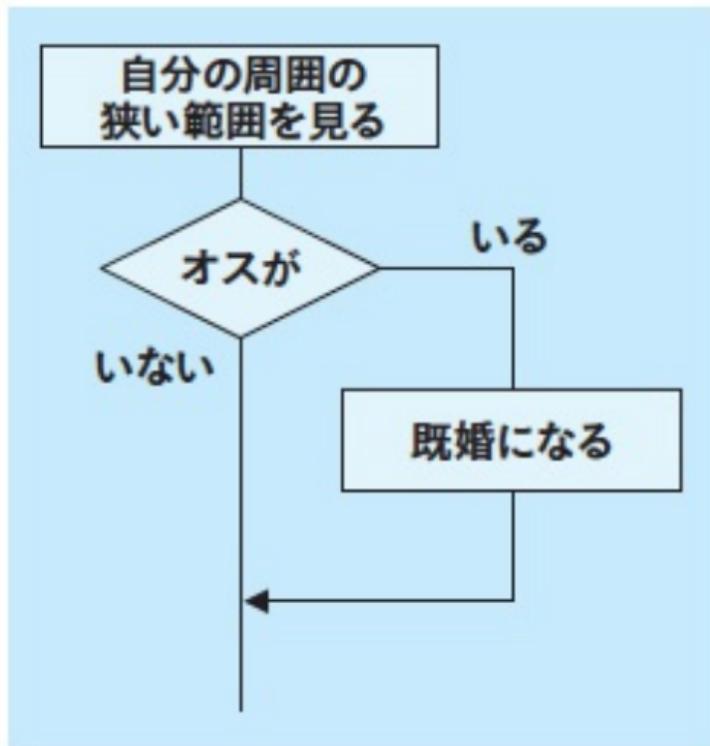


図14.2

未婚メスはオスがすぐそばに寄ってくれれば既婚状態になります。

以下のルール表記では、狭い視野(1)で周囲のエージェントをさがし、オスがいれば結婚(青色に変化)します。

```
MakeAllAgtsetAroundOwn(neighbor, 1, False)
For each one in neighbor
    If one.condition == Color_Red Or one.condition == Color_Yellow
    Then
        My.condition = Color_Blue
    End if
Next one
```

ここでは、前章で学んだフォ・イーチ文が使われています。新しい技法は含まれていません。

なお、既婚メスは何もしませんから、これで全てです。

14.5 オスのルール

オスの行動ルールはメスと比べて複雑です、次のようにになります。

未婚オスは周囲を広く調べて、チョウが全くいなければ、ヒラヒラと飛び回ります。チョウが見つかれば(2匹以上見つかると、ランダムに1匹だけ選び出して)、そこに飛んでいき、未婚メスかどうか調べます。もし未婚メスならそこにとどまって結婚し、それ以外なら飛び去ります。

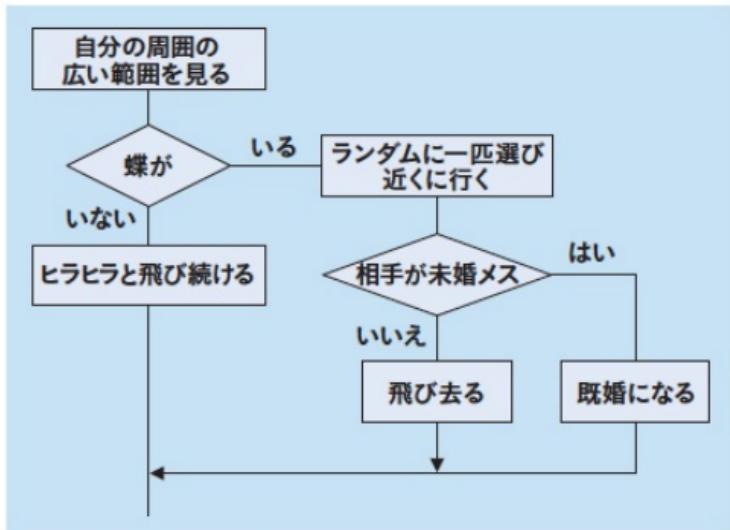


図14.3

```

MakeAllAgtsetAroundOwn(neighbor, 2, False)
If CountAgtset(neighbor) == 0 Then //周囲にチョウがない場合
  Turn(Rnd() * 60 - 30)
  Forward(0.6)
Else //チョウがいる場合
  one = GetAgt(neighbor, Int(Rnd() * CountAgtset(neighbor)))
  My.X = one.X + 0.5
  My.Y = one.Y
  If one.condition == Color_Magenta Then //未婚メスの場合
    My.condition = Color_Yellow
  Else //未婚メスではなかったら
    Turn(Rnd() * 60 - 30)
    Forward(2)
  End if
End if
  
```

上のルール表記で

```
one = GetAgt(neighbor, Int(Rnd() * CountAgtset(neighbor)))
```

が特定のエージェントを選び出す技法です。

ここで初めて登場した関数GetAgt(S, m)は、エージェント集合型変数Sにしまわれているエージェントから、m番目のエージェントを選び出すルールです。そして選ばれたエージェントは、one(エージェント型変数)に代入されます。

上の例では、CountAgtset(neighor)は1以上の場合に限られますから(冒頭のイフ文で0の場合を除外しています)、GetAgt(neighor, Int(Rnd() * CountAgtset(neighor)))は、neighorの中にいるエージェント数(CountAgtset(neighor))に応じて、ランダムに1つ選び出すことを意味しています。

なお、エージェント集合型変数neighorの中にn個のエージェントがリストアップされているとき、GetAgt(neighor, 0), GetAgt(neighor, n - 1)などと書き込むと、neighorの中の最初のエージェントや最後のエージェントが選ばれます。ただし、neighorにエージェントを取り込んだときに何が最初で何が最後になるのかは意外と難しい問題です(コラム「[エージェント集合の中のエージェントの並び方](#)」を参照)。さしあたり、存在しない番号を指定することのないようなルールにするように気を付けて下さい。特に、neighorにエージェントが全くリストアップされていないのにGetAgt()を使ってエージェントを取り出そうとするとエラーになるので、常にエージェント数がゼロの場合を除外するクセをつけて下さい。

ルールをルールエディタに全て書き込んだら、mating(B)という名前を付けて保存してから実行して下さい([図14.4](#)参照)。チョウの動き(もっともオスしか動いてませんが)にも注目しましょう。本物らしく見えるでしょうか? パソコンの性能にもよりますが、実行ウェイトを数十ミリ秒にすると、チョウの飛翔のように見えます。

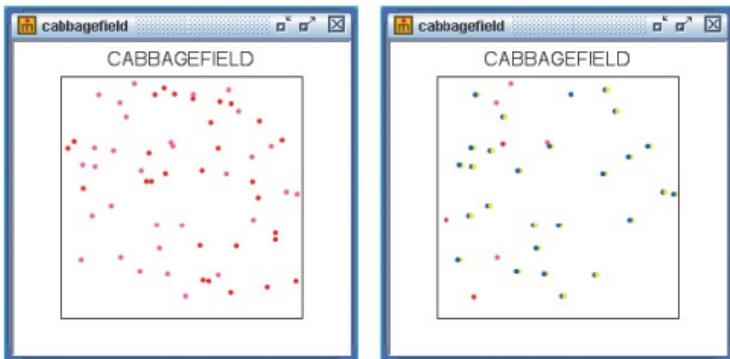


図14.4

新しく学んだ事項

- エージェント集合型変数の中からエージェント1体を選び出す
- GetAgt()

練習問題

次のような事例を考えましょう。

広告メディアが互いに異なる製品を宣伝している。宣伝に接した消費者の一部はその製品を使うようになる。

まず、Universeの下に空間marketを作り(設定はデフォルト値のまま)、その下にmediaエージェントを10、consumerエージェントを100作ります。mediaにはproduct、consumerにはfavoriteをどちらも整数型の変数として作ります。マップ出力では、両タイプのエージェントも追加した変数を指定する色表示とします。

広告メディアは、最初に、アカ製品を宣伝するメディア（赤色表示）とアオ製品を宣伝するメディア（青色表示）を半々の確率で存在するようにします。

```
Agt_Init{  
My.X = Rnd() * 50  
My.Y = Rnd() * 50  
If Rnd() >= 0.5 Then  
    My.product = Color_Red  
Else  
    My.product = Color_Blue  
End if  
}
```

他方、消費者も最初、marketにランダムに存在しています。

```
Agt_Init{  
My.X = Rnd() * 50  
My.Y = Rnd() * 50  
My.Direction = Rnd() * 360  
}
```

毎ステップ、メディアは場所を変えずに宣伝し続けるのに対し、消費者はmarketの中を動き回ります。

これからが問題です。

14.1

消費者は、身近な（視野範囲が2）メディアから影響を受け、3割の確率でそのメディアが宣伝している製品を好むようになる。

視野内にメディアがいる場合、自分が影響を受けるメディアはランダムに選ばれることにしましょう。



GetAgt()を使う例です。

14.2

消費者は、身近な（視野範囲が2）メディア（複数いる場合はランダムに選択）が宣伝している製品と同じ製品を使っている他の消費者が身近（視野範囲が2）に複数いる場合にかぎり、その製品を好むようになる。



フォ・イーチ文も使います。

第15章

他のエージェントに働きかける

15.0 他者の状態を変える

- 周囲のエージェントの属性(内部状態)や行動を変えられます
- 他のエージェントの変数値を操作する方法を学びます
- GetAgt()の応用です
- エージェントの内部状態を論理的な真偽で区別します
- 牧羊犬のモデルを作ります

15.1 積極的な働きかけ

マルチエージェント・シミュレーションでは、エージェントが周囲から影響を受けるだけでなく、周囲に影響を与えることも重要です。

第13章でモデルを作った流行現象は、病人が健康な人に病気をうつすという意味では、エージェントが周囲に影響を及ぼしています。しかし、病人は(悪意がないかぎり)意図的に他人に病気をうつそうと行動しているわけではありません。そこで、モデル化の基本的な考え方は、健康な人が周囲にいる病人から影響を受ける、というものでした。それに対し、この章では、自分が他者に意図的に働きかける(作用する)場合を想定します。

他のエージェントに作用するモデルの基本は、

働きかける側のエージェントには

1. 周囲を見回して、周囲にいるエージェントを認識する。

2. 彼ら(の全員または一部)の属性を変える。

働きかけられたエージェントには

1. 属性に応じた行動のレパートリーを持っている。
2. 他のエージェントに変えられた属性に対応する行動をとる。

というルールを記述することです。artisocの文法としては今まで学んだ内容でルール化することができますが、新しい技法が登場します。

15.2 牧羊犬の仕事をモデル化する

あるエージェント種に属すエージェントが別のエージェント種に属すエージェントに働きかけるモデルを作ります。

牧場では夕方、飼っている羊を柵に入れなければならない。牧羊犬が羊を追い込む仕事をする。

では、モデル作りの準備作業に入りましょう。

1. Universeの下に空間meadowを作ります。設定は、デフォルト値のままにしておきます。
2. meadowの下に、sheepを100匹、dogを10匹作ります。
3. sheepには牧羊犬に捕まったかどうかを表す布尔型変数caughtを作ります。布尔型変数は、TrueかFalseのどちらかの値をとります。ルールの中では、捕まったらTrue、そうでなければFalseにしましょう。
4. 出力設定で、マップ出力を設定します。
5. モデルをshepherdという名前で保存して下さい。

モデル化に際しては、エージェントの行動の本質的な部分だけ取り出して、次のように考えましょう。

1. 羊は、柵の外ではうろうろと歩き回る。なお、柵の内側はmeadowのX座標が0から10、Y座標が0

から10の範囲の正方形とし、柵の外はそれ以外とします。

2. 牧羊犬に捕まった羊は、柵に追い込まれる(ルールを単純にするため、瞬間移動で柵の中に入る)。
3. 牧羊犬は牧場を走り回り、羊がそばにいると、羊を1頭ずつ捕まえる

15.3 羊たちの行動

牧場の羊たちの行動ルールは次のようにになります。

フローチャートに対応する具体的なルールを書きましょう。sheepのルールには新しい技法は含まれていません。

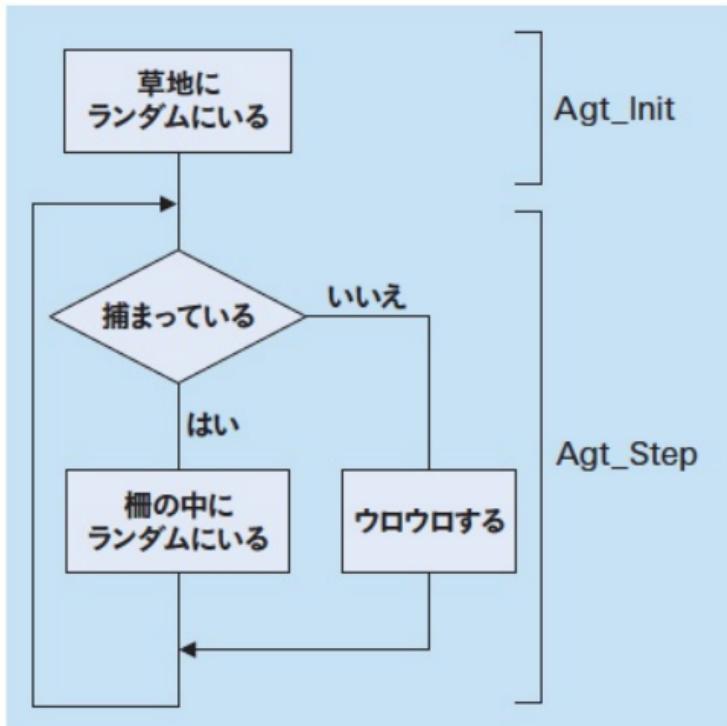


図15.1

- 最初は、まだ牧羊犬に捕まっておらず、草地にランダムにいます。

```

Agt_Init{
  My.X = Rnd() * 50
  My.Y = Rnd() * 50
  My.Direction = Rnd() * 360
  My.caught = False
}
  
```

2. 每ステップ、捕まつたかそうでないかで、異なる行動をとります。

```
Agt_Step{
If My.caught == True Then //捕まつたら
    My.X = Rnd() * 10
    My.Y = Rnd() * 10
    My.caught = False //状態を元に戻す
Elseif My.X >= 10 Or My.Y >= 10 Then //柵の外にいるなら
    Forward(0.5)
    Turn(Rnd() * 40 - 20)
End if
}
```

15.4 牧羊犬の行動

牧羊犬の行動は図15.2のフローチャートが示すように、単純です。しかし、ルールには新しい技法が使われています。書き込むべきルールを紹介した後で、新しい技法を詳しく説明します。

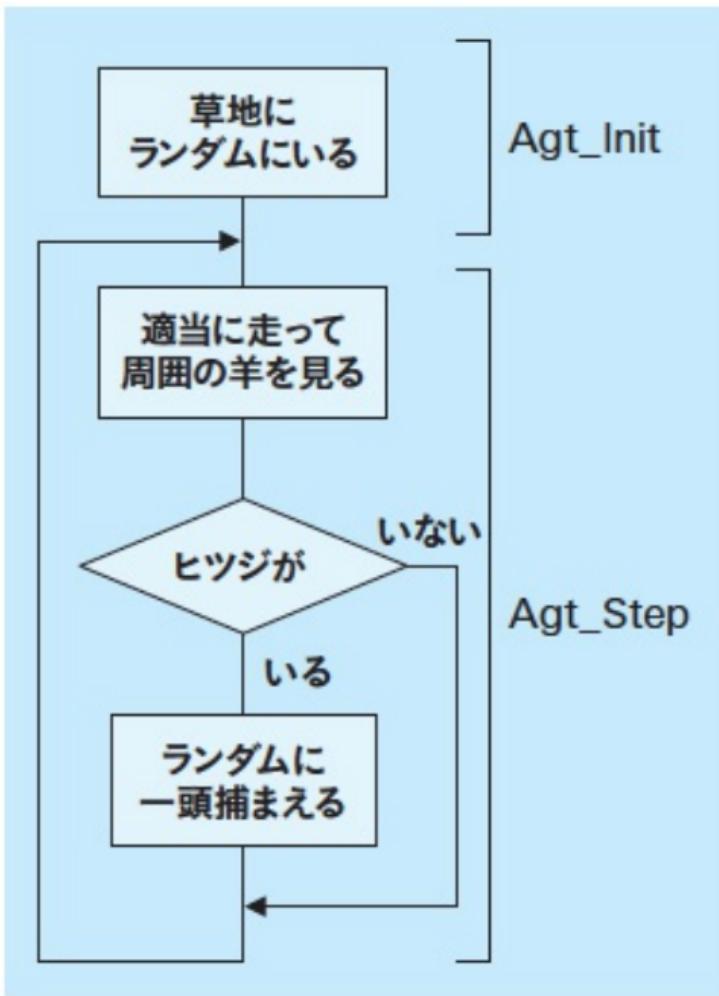


図15.2

1. 初期状態は、羊と同様です。dogエージェントのルールエディタの中のAgt_Init{}に書き込んで下

さい。ルールの説明は省略します。

2. 牧羊犬は毎ステップ走り回って、羊をさがし、近くにいると、1頭を捕まえます。

```
Agt_Step{
Dim neighbor As Agtset
Dim target As Agt
Dim num As Integer
Forward(1)
Turn(Rnd() * 40 - 20)
MakeOneAgtsetAroundOwn(neighbor, 2, Universe.meadow.sheep,
False)
num = CountAgtset(neighbor)
If num >= 1 Then //近くに羊があれば
    target = GetAgt(neighbor, Int(Rnd() * num))
    target.caught = True
End if
}
```

後半の4行が新しい技法です。

後半で使う一時的変数targetはルールエディタの冒頭で宣言してあります。まず、MakeOneAgtsetAroundOwn()で視野2の範囲で周囲に羊がいるかどうか調べ、neighborにリストアップします。次にneighborにリストアップされているsheepが何頭いるかCountAgtset()で調べます。もし1頭以上いれば、そこからランダムに1頭選んで、targetという一時的なエージェント型変数にします。そしてその1頭を、捕まつた状態に変えます(target.caught = True)。

第14章では、選ばれたエージェントの属性を調べて、それによって自分の行動を変えました。ここでは、選ばれたエージェントの属性を、代入文を使うことによって、直接変えています(target.caught = True)。ここが従来と大きく異なる点です。

それでは上書き保存してから実行してみましょう([図15.3](#))。

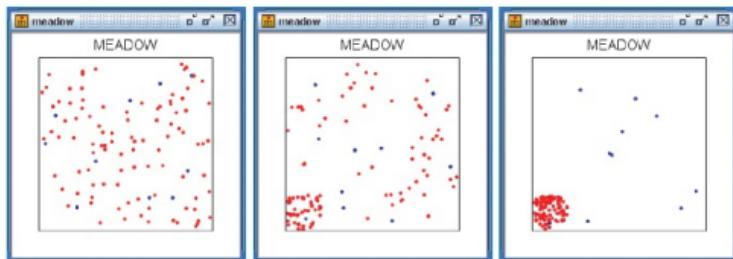


図15.3

15.5 作業を完了する時間を調べる

他のエージェントに作用するモデルの基本は以上です。次に、このモデルを使って、「牧場経営」に役立ててみましょう。

羊を飼うには牧羊犬が必要だが、牧羊犬には優秀なのもいれば、駄犬もいる。それでは高価で優秀な牧羊犬を少数入手すべきなのか、それとも廉価で平凡な牧羊犬を多く入手すべきなのか。

エージェントのプロパティを開いて、dogの数を変化させながら、同時にエージェントのルールエディタの中のMakeOneAgtsetAroundOwn()の中の視野の広さやForward()の中の数値(足の速さ)を変化させて、羊を全て柵の中に追い込むまでの時間を比べてみましょう。

もちろん、プロパティ・ウインドウやルールエディタを開いていちいち操作するのではなく、最初にコントロールパネルを作つてから実験をしても構いません。コントロールパネルの作り方や、必要な修正作業は、第7章で学びましたね。

そこで、柵の中にいる羊を数える作業をUniverseの中でしましょう。集計するために、Universeに整数型変数infenceを作ります。集計作業とシミュレーション終了のメッセージ出力は、既に学んだ技法ですが、念のため、下記に書き出します。

```

Univ_Step_End{
Dim total As Agtset
Dim one As Agt
Universe.infence = 0 //初期化
MakeAgtset(total, Universe.meadow.sheep) //全ての羊をtotalに格納
For each one in total
  If one.X < 10 And one.Y < 10 Then //柵内にいれば、カウントする
    Universe.infence = Universe.infence + 1
  End if
Next one

If Universe.infence >= 100 Then
  ExitSimulationMsgLn("Completed after" & GetCountStep() &
"steps")
End if
}

```

shepherd(B)という名前を付けて保存し、実行してみましょう。牧羊犬の数と牧羊犬の能力(視野の広さや足の速さ)を変えることで、シミュレーションが終了するまでのステップ数が変わる様子を見て取れたと思います。

視野の広さ、足の速さ、頭数の違いが、シミュレーション終了ステップ数にどのように影響を与えるか、図表を書いてみましょう。

もちろん、有能な牧羊犬を多く使うと、すぐに柵の中に羊を追い込めますが、なにしろ高価です。能力の違う牧羊犬の価格を適当に決めて、なるべく速く作業を終えるという目的と牧羊犬を買うための資金の制約とから、最適な牧羊犬の買い方を考えてみましょう。

15.6 モデル作りを工夫する(とばしてもかまいません。)

上のshepherd, shepherd(B)モデルは、牧羊犬が羊を柵に追い込むという作業の必要最低限の本質を取り出してモデル化したものです。そこで、少し工夫を加えることにしましょう。

牧羊犬は柵の外だけを走り回る。

上のモデルでは、牧羊犬は柵の中にも入って作業をします。そのような無駄を省くには、たとえば、次のよ

うなルールを加えればよいでしょう。どこに挿入すれば良いか、よく考えてから書き加えて下さい。

```
If My.X < 10 And My.Y < 10 Then
  If Rnd() >= 0.5 Then
    My.X = 11
    My.Y = Rnd() * 11
  Else
    My.X = Rnd() * 11
    My.Y = 11
  End if
End if
```

このモデルは、shepherd(C)という名前を付けて保存して下さい。

他にも、いろいろと工夫してみて下さい。

もっと複雑な行動については第21章で学びます。

新しく学んだ事項

- 他エージェントの変数を変えることにより、そのエージェントに自分の行動を変えさせる
- ブール型変数とその使い方

練習問題

次のような事例を考えましょう。

広告メディアが互いに異なる製品を宣伝している。宣伝に接した消費者の一定割合にその製品を使うようにさせる。

これは第14章の練習問題と全く同じ設定にしましょう。

ここからが問題です。モデルとしては、消費者の好みを変えるだけで、実際にその製品を使うという行動ルールを書く必要はありません。

15.1

メディアは、近くにいる消費者の中からランダムに選ばれた一人の好みを宣伝している商品に変える。



牧羊犬モデルと同構造, target.favorite = My.product

15.2

メディアは近くにいる全ての消費者に影響を及ぼし、3割の確率で消費者は宣伝している製品を好むようになる。



フォ・イーチ文, target.favorite = My.product

働きかけは「禁じ手」か「一の手」か

マルチエージェント・シミュレーションでは、エージェントどうしの相互作用が重要な位置を占めています。しかし、エージェントは他のエージェントの何に作用するのでしょうか。作用を受けたエージェントは何が変わるのでしょうか。

エージェントの属性は、誰が変えられるのか、という問題を考えましょう。エージェント自身だけなのか、他者にも可能なのか。これはマルチエージェント・シミュレーションの原理にも関わる難問なのです。

穏当な解答は、エージェントは自律的のだから、属性はエージェント自身のみが変えることができ、他の誰にも変えることはできない、というものでしょう。artisocの技法に即していえば、「My.変数 =」という代入文のみが文法的に許されている場合です。周囲にいるエージェントから1体をtargetとして選び出して、「Target.color = My.color」などという「相手を自分の色に染め上げる」ことは許されないことになります。

こうしておくと、各ステップ、エージェントは自分の番が来たときだけ自分の属性を変えることができるので、シミュレーションの流れは単純で、管理しやすくなります。しかし、この方式は技術的には楽ですが、エージェントどうしを相互作用させようとすると不便すぎます。

エージェントPがAをしているエージェントQにBをさせる、というモデルを作る場合を考えましょう。まず、QにはPからのメッセージ(命令、要請、懇願)を受けるセンサー(メッセージに応じて値が変わる変数)が必要です。さらに、Bは現在の行動(A)とともにQの行動ルールのなかに書き込まれている必要があります。

もし自分自身でしか属性を変えられないとすると、誰か自分に何かをさせたいのか、毎ステップ、全て(あるいは周囲)の他エージェントに「ご用聞き」をしなければ(フォーメーチ文の利用)、センサーの状態を変えられません。Pが自分(Q)に対して「Bをせよ」というメッセージを出しているとセンサーが反応して、初めて、QがAをやめてBをする可能性が出てきます。

もし、Pが直接Qのセンサーに作用する、という技法が許されれば便利になります。PがQのセンサーの

状態(属性)を変えることができれば、その結果、QがAをやめて、その代わりにBという行動を実際にとるというプロセスが実現します。たとえば、QのルールがMy.sensorB の値がTrueのときにはBをするようになつていれば、PがQをtargetとして選び出してセンサーに作用する(Target.sensorB = True)と、QはBをするようになります。このようなルールを書くことは、artisocで簡単にできます。

しかし、便利になるだけでなく、問題も出てきます。

たとえば、あるステップで、PがQのセンサーに作用してからQの行動ルールが実行されれば、ただちにQはBをしますが、Qの後にPの行動ルールが実行されると、QがBをするのは次のステップになってしまいます。つまり、シミュレーションの最中の「出来事」を管理することが難しくなるのです。これについては、第19章や第31章を参照して下さい。

エージェントへの究極の働きかけは「抹消」ですが、これにも問題があります。あるステップで、QがPを消してしまったと仮定しましょう(DelAgt()の実行)。ところが他のエージェントはPが消えてしまったことを知らされていません。その後に、別のエージェントがPに何か作用しようとしても、Pはもはや存在していません。どうしたらよいのでしょうか。困るのは私たちだけではありません。artisocも困ってしまい、場合によってはエラーメッセージを出すかも知れません。これに関してはコラム[「エージェントを消す三つの関数」](#)を参照して下さい。

このように、他のエージェントへの働きかけのルールは便利なだけに、ルールを記述する際には、「第三者への迷惑」が生じないか、十分に思考実験をすることをお勧めします。

第16章

エージェントへの究極の働きかけ

16.0 エージェントそのものに作用できます

- エージェントに対する究極の働きかけは、エージェントの生成と抹消です
- ブランクトンの食物連鎖をモデル化しましょう
- シミュレーションの最中にエージェントを作ります
- シミュレーションの最中にエージェントを消します
- 新しく生成されたエージェントの変数に値を設定します
- 自分で自分を消すことも可能です

16.1 エージェント自体への働きかけが可能です

他のエージェントに対する働きかけの極端なケースは対象エージェントを消してしまったり、逆に新しいエージェントを作ったり、というものでしょう。比喩的に言えば、エージェントがエージェントを殺す、エージェントを産む、という行為ですね。もちろん、働きかけられるエージェントの立場に立って、エージェントが死ぬ、エージェントが生まれる、と置き換えて考えることもできるでしょう。

artisocでは、最初にエージェントを作って配置するだけでなく、シミュレーションの途中でもエージェントを作ることができます。また、エージェントを殺す（エージェントが死ぬ）という場合でも、単にエージェントが動かなくなるとか、他から影響を受けなくなるというだけでなく、モデルの中から「抹殺（抹消）」してしまうこともできます。この章では、シミュレーションの過程で、どのようにして、エージェントを消したり作ったりするのか、という技法を学びます。

なお、消すルールの応用として、自分で自分を殺す（自殺する）ことも可能です。

16.2 「殺す」は「消す」

artisocでエージェントを死なせる操作は、すでに第13章の練習問題でinfectionモデルの修正という形で学んでいます。その方法は、患者の状態を黒色にして、動かなくなる、そしてこの状態になったエージェントから他のエージェントは影響を受けないようにする、というものでした。つまり、そこでの死亡とは、モデルの中でエージェントが何からも影響されず、何にも影響しない状態にすることです。それに対してここで学ぶのは、特定のエージェントをモデルの中から消し去る特別な方法です。

ここで次のようなモデルを作りましょう。

水槽に植物プランクトンと動物プランクトンが浮遊しており、後者が前者を餌にしている。

それでは、aquariumという空間（デフォルト）に植物プランクトンphyto（phytoplanktonの略）を200、動物プランクトンzoo（zooplanktonの略）を20作ります。適当にマップ出力を設定して下さい。モデル名はplanktonにして保存しましょう。

phytoプランクトンの行動ルールは簡単で、浮遊するだけです。例によってランダムな位置と向きに初期配置し、プラスマイナス60度の範囲で向きを変えながら、毎ステップ0.5進みます。このルールはもうおなじみなので、ルール記述は省略します（ルールの書き込みに自信のない人は、下記のzooプランクトンのルールの最初の数行を参考にして下さい）。

zooプランクトンはphytoプランクトンと同じような行動ルールですが、プラスマイナス30度の範囲で向きを変えながら、毎ステップ2進むものとします。さて、zooはphytoを餌にします。周り（視野2の範囲）にphytoがいれば1ステップに1匹食べることにしましょう。「食べる」という行動を、ここではphytoを1匹消すというルールで表すことにします。周りを見回して、存在しているphytoの集合をしまうエージェント集合型の一時的変数surroundと餌となるphytoに対応するエージェント型の一時的変数foodを最初に宣言しておきます。zooエージェントのルールは次のようになります。

```
Agt_Init{
```

```

My.X = Rnd() * 50
My.Y = Rnd() * 50
My.Direction = Rnd() * 360
}
Agt_Step{
Dim_surrond As Agtset
Dim_food As Agt
//動き回る基本パターン
Turn(Rnd() * 60 - 30)
Forward(2)
//餌をさがして、あれば食べる
MakeOneAgtsetAroundOwn(surrond, 2, Universe.aquarium.phyto,
False)
If CountAgtset(surrond) > 0 Then
  food = GetAgt(surrond, Int(Rnd() * CountAgtset (surrond)))
  DelAgt(food) //食べる
End if
}

```

GetAgt()という組み込み関数、GetAgt()で得られたエージェントをエージェント型変数に代入するルールは第14章で学びました。

DelAgt()は新しく学ぶルールです。具体的には、DelAgt(food)というルールでfoodを消すわけですが、モデルから実際に消えるのはfoodにさせられたphytoエージェント1個です。

それでは、上書き保存してから実行ボタンを押して、動物プランクトンが植物プランクトンを食べていく様子を観察して下さい([図16.1](#))。

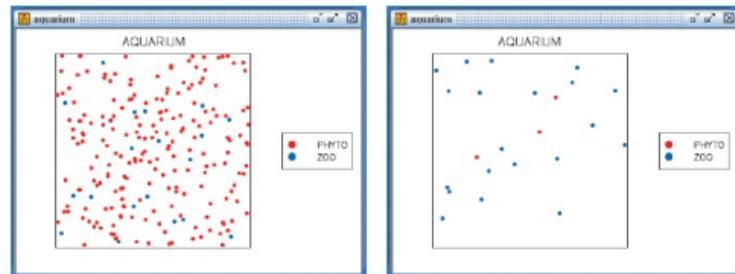


図16.1

16.3 「産む」は「創る」

上のモデルでは、植物プランクトンはやがて食べ尽くされてしまいます。そこで、植物プランクトンは適当な条件下で繁殖するように、モデルを修正しましょう。

植物プランクトンは増殖する。

エージェントが別のエージェントを生み出す、という行動はCreateAgt()でルール化します。第7章で、エージェント数をコントロールパネルで初期設定し、その数だけのエージェントをUniverseのルールエディタの中で作る技法を学びました。そのときに用いたCreateAgt()と同じです。それを、phytoエージェントのルールエディタの中で使うのです。なお、薄字のルールは既に記入済みのはずです。

```
Agt_Step{  
Dim daughter As Agt //生成されるエージェントの「仮の姿」  
Turn(Rnd() * 120 - 60)  
Forward(0.5)  
If Rnd() < 0.05 Then  
    daughter = CreateAgt(Universe.aquarium.phyto)  
    daughter.X = My.X  
    daughter.Y = My.Y  
    daughter.Direction = Rnd() * 360  
End if  
}
```

上のルールでは、5%の確率でCreateAgt()によってphytoエージェント1個を新しく作り、そのまま一時的なエージェント型変数daughterにします。そして、できたてのエージェントに位置や向きを設定します。形式的にはdaughterの変数を設定していますが、daughterの中身は実質的にはphytoエージェントです。この例では、「親」と同じ場所にし、向きはランダムに設定しております。

なお、CreateAgt()を実行すると、ただちにそのエージェント種のAgt_Init{}のルールが実行されます。このモデルの場合は、phytoのルールに従ってdaughterの位置も向きもランダムに設定されますが、その後で、CreateAgt()に続く代入文で、親と同じ位置に設定され直されています。言い換えると、向きの設定は冗長(不要)です。

phytoエージェンが増殖するルールを追加したモデルを、plankton(B)という名前を付けて保存して下さい。それでは実行してみましょう。

植物プランクトンがだんだん増えていくはずです(図16.2参照)。そのうち、植物プランクトンでいっぱいになりますが、エージェント数が増えることによってパソコンにかかる負荷が大きくなってしまいます。だんだん実行するスピードが落ちるので、適当なところで停止ボタンを押して下さい。最悪の場合は、シミュレーションを停止する前にフリーズしてしまい、再起動を余儀なくされるからです(保存していないファイルは消滅します)。

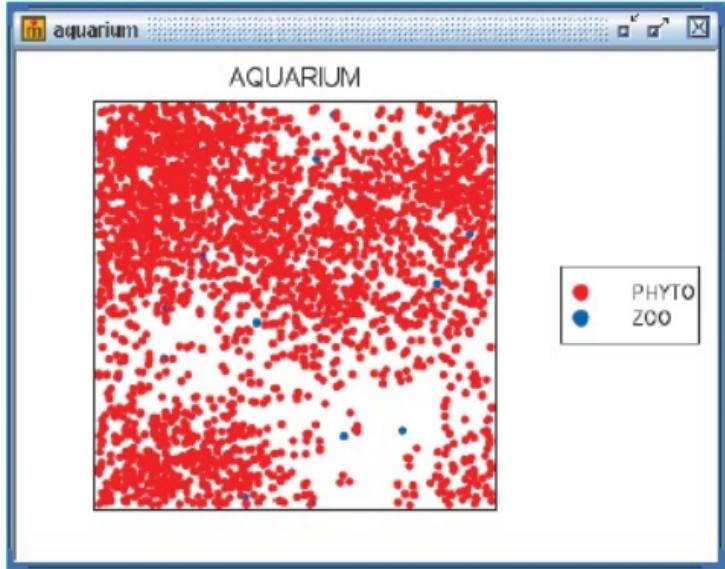


図16.2

16.4 個体数変動を調べる

プランクトンの数を時系列グラフに出力しましょう。また、プランクトンがいなくなったらシミュレーションを自

動終了させましょう。今のところ、動物プランクトンは一定数で、植物プランクトンしか変動しませんが、16.6で動物プランクトンも変動するルールが登場するので、どちらかのプランクトン数がゼロになったら終了するルールにします。

たとえば、Universeの下に整数型変数numphytoとnumzooを追加して、次のようなルールをUniverseのルールエディタの中に書き込みます。

```
Univ_Step_End{
    Universe.numphyto = CountAgt(Universe.aquarium.phyto)
    Universe.numzoo = CountAgt(Universe.aquarium.zoo)
    If Universe.numphyto <= 0 Or Universe.numzoo <= 0 Then
        ExitSimulationMsgLn("Extinct after" & GetCountStep() &
        "steps")
    End if
}
```

上のルールで、CountAgt()は初出です。これは、指定されたエージェント種の数を数える関数です。エージェント集合型変数にリストアップされているエージェントの数を数えるCountAgtset()と混同しないようにしましょう。

numphytoとnumzooの時系列グラフへの出力方法は自分で工夫して下さい。2種類のプランクトンの数は桁が違うので、たとえばnumphyto / 10とすると同じようなスケールになるはずです。

プランクトン数を集計するルールを加えたモデルは、plankton(C)という名前を付けて保存して下さい。実行すると、[図16.3](#)のような時系列グラフが出るはずです。

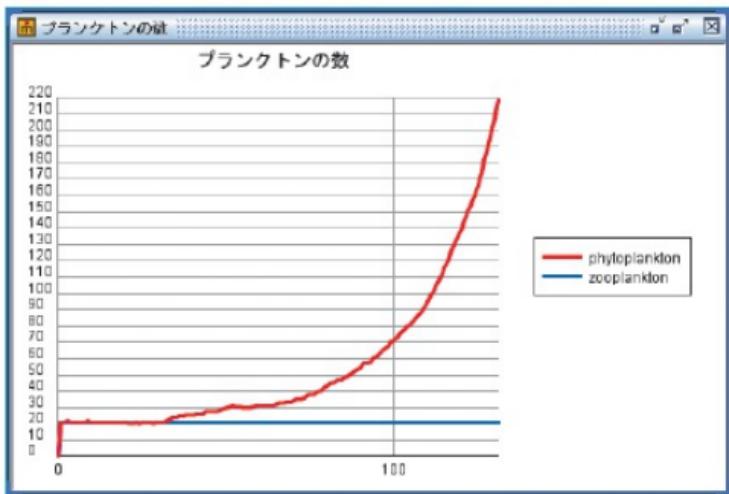


図16.3

CountAgt()で、生きているプランクトンの個体数を調べることができるのは、死んだプランクトンが「消えてしまった」からです。DelAgt()の効果がここに現れています。エージェントに生死を区別する変数(たとえばcolor)を追加して、その変数の値(たとえば色)で生死を判別する場合とはルール記述が違うことがわかると思います。

16.5 過ぎたるは及ばざるがごとし

上で作ったplanktonモデルでは、phytoエージェントは無限に自己増殖しますが、これは不自然な仮定です。そこで、増えすぎると死ぬように修正してみましょう。

植物プランクトンが周囲に増えすぎると環境が劣化して、自分自身が死んでしまう。

次のルールを付け加えましょう。

```

Agt_Step{
Dim surround As Agtset
Dim daughter As Agt
(中略)
End if
MakeOneAgtsetAroundOwn(surround,1,Universe.aquarium.phyto,
False)
If CountAgtset(surround) >= 4 Then
  DelAgt(My)
End if
}

```

周囲(視野1の範囲)にphytoが4個以上あると酸素不足か栄養不足かはさておき、自分が死んでしまいます。DelAgt(My)が自分を消す方法です。DelAgt()の中には、エージェント型の変数を入れますが、Myと入れることで自分を指定しています。

こうすることにより、植物プランクトンの爆発的発生は生じなくなります。

それでは、plankton(D)という名前で保存してから、実行しましょう。(図16.4参照)

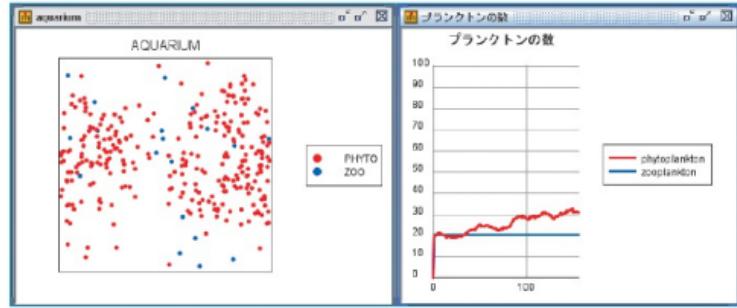


図16.4

16.6 自然の生態系に近いモデルをめざして

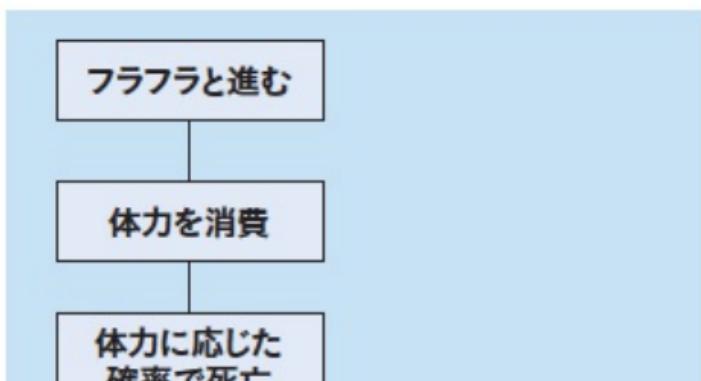
この章で学んだ方法を活用して、planktonモデルをもう少し自然の生態系に近づけてみましょう。

海洋に植物プランクトンと動物プランクトンが浮遊しており、後者が前者を餌にしている。植物プランクトンは増殖するが、増えすぎると死んでしまう。動物プランクトンは、周囲の植物プランクトンを食べる。餌を食べないと体力が消耗してやがて死んでしまうが、たくさん食べると増殖する。

植物プランクトンについては以上で作ったモデルのルールそのままにしましょう。動物プランクトンについては、少し追加作業が必要です。

1. 体力を表すpowerという整数型変数を追加する。
2. 最初4から9の間の体力(power)をランダムで与える。
3. 体力は毎ステップ1ずつ減少するが、植物プランクトンを1個食べると3上昇する。
4. 体力が減るとそれに応じて死ぬ確率は高くなる(ゼロなら必ず死に、1ならば20%、10ならば2%の確率で死ぬ)。
5. 体力が10を超えると増殖するが、「親」プランクトンは体力を4減らし、「子」プランクトンは体力4を与えられて生まれる。
6. 「子」プランクトンは、「親」と同じ位置に生まれるが、向きはランダムとする。

[図16.5](#)のフローチャートのようなzooプランクトンの行動ルールは以下のようになります。薄字のルールは、既に記入済みです。



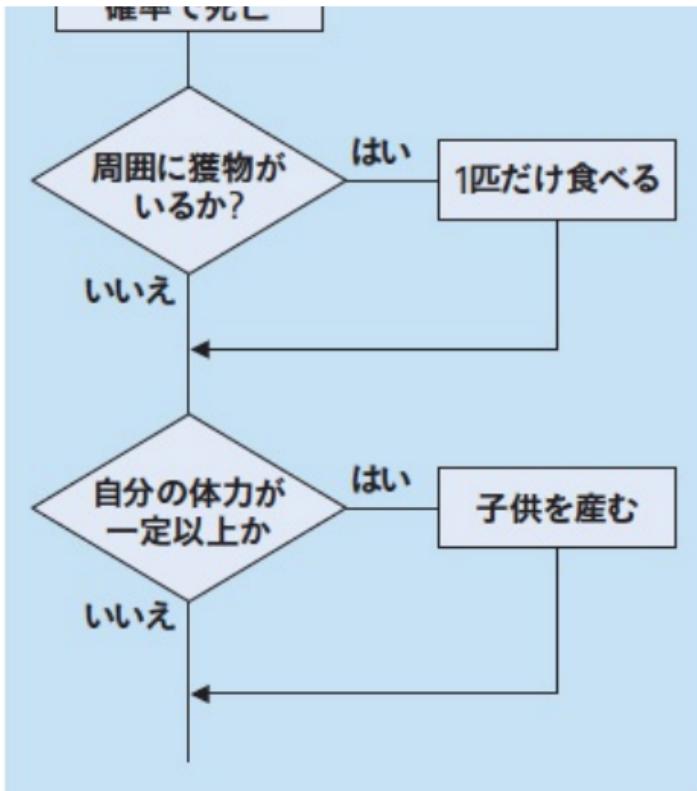


図16.5

```

Agt_Step{
Dim surround As Agtset
Dim food As Agt
Dim daughter As Agt
Turn(Rnd() * 60 - 30)
Forward(2)
My.power = My.power - 1
If Rnd() * My.power <= 0.2 Then
  DelAgt(My)
End if
MakeOneAgtsetAroundOwn(surround, 2, Universe.aquarium.phyto,
False)
  
```

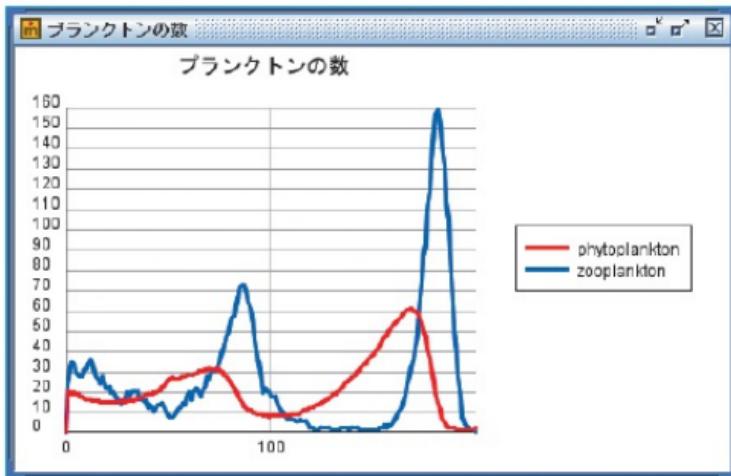
```

If CountAgtset(surround) > 0 Then
    food = GetAgt(surround, Int(Rnd() * CountAgtset(surround)))
    DelAgt(food)
    My.power = My.power + 3
End if
If My.power >= 10 Then
    daughter = CreateAgt(Universe.aquarium.zoo)
    daughter.X = My.X
    daughter.Y = My.Y
    daughter.Direction = Rnd() * 360
    daughter.power = 4
    My.power = My.power - 4
End if
}

```

上のルール表記で新しく登場した文法や技法はありません。ルールを読みとれますか。それは、plankton(E)という名前で保存して、実行してみましょう。

このモデルはプランクトンを想定していますが、本質的には捕食者と被捕食者の相互作用のモデルですから、餌が多いと捕食者が増え、その結果餌が減って捕食者も減り、すると餌が増え始める、という循環的な特徴を持っているはずです。しかし、なかなか安定的に循環しません([図16.6](#)参照)。それは、餌も捕食者も空間的に散らばっているために、空間を考慮しない数理モデルのようにきれいな結果にならないからです。むしろ、このモデルの方が現実に近いのではないでしょうか。実際、餌が遠くにあっても食べられないわけですから、餌がなくなる前に、捕食者が絶滅してしまう可能性の方がずっと大きいのです。要するに、捕食者と被捕食者との間に循環的な関係が形成されるのは、それほど簡単なわけではありません。



新しく学んだ事項

- CreateAgt()をエージェントの行動ルールとして用いる
- CountAgt()
- DelAgt()
- DelAgt(My)
- エージェント数をあまり多くしない

練習問題

16.1

CreateAgt(), DelAgt()を使う簡単な問題である。

牧場に生えている草を羊が食べる。

kusaharaという空間にkusaとhitsujiエージェントを追加し, kusaが増殖するルールと羊が草を食べるルールを書きましょう。

16.2

planktonモデルに, zooプランクトンを食べるfishエージェントを追加する。



fishの数は少なくしましょう。

生態系に関心のある人は, 次の作業を試みるのも面白いでしょう。

16.3

plankton(E)モデルには, phytoが増えたり死んだりする条件, zooがphytoを食べて増える体力, 每ステップの体力消耗度, 体力と死亡率との関係, 増殖する際の体力の親から子への移動量など, たくさんのパラメータがある。それらを適当に調整して, 安定的な(周期的にプランクトン数が変動する)状態を作り出してみよう。

第17章

非対称的な相互作用を複雑化する

17.0 そう簡単には従わない

- Let'sとDon'tの効果はいかに?
- 思いのままにならないところが面白いのです
- 影響を及ぼす側と及ぼされる側との関係をモデル化します
- 誘導のモデルと取り締まりのモデルを作ります

17.1 一方的関係から非対称的かつ双方的な関係へ

第15章では、他のエージェントに働きかける技法として、(1)自分のエージェント型変数に他のエージェント取り込んで、(2)そのエージェントの変数の値を変えてしまう技法を学びました。この章では、その技法と第14章で学んだ周囲のエージェントを調べて選び出す技法を駆使して、働きかけとそれへの反応との間の複雑な関係をモデル化する技法を学びます。

他のエージェントへの作用にはある行動をさせる場合と、現在とっている行動を禁止する場合とがあります。しかし、単に特定の行動をやらせたり、やめさせたりすることは、影響力の行使としては単純です。なぜなら、相手はこちらの命令通りに動いているだけだからです。artisocの技法としては、相手の状態を変えて、特定の行動をとらないようにしたり、逆に特定の行動をとるようにしたりするだけです。

そこで、もう少し複雑な状況を想定しましょう。ここでは、相手に特定の行動をさせようとするがなかなか言うことを聽かない場合と、特定の行動を禁止しようとするが相手がある程度逆らって言うことをきかない場合をモデル化します。働きかけに対してどのような逸脱があり得るのか、がポイントです。新しい技法が登場するわけではありませんが、今まで習ったことを組み合わせます。

17.2 幼稚園の先生の苦労をモデル化する

まず、自分が周囲の人たちに積極的に働きかけて、同調させるモデルを作りましょう。

大きな幼稚園で、子供たちが歩き回っている。先生は、みんなを東向きに歩かせようとして、あちこち走り回りながら、自分の周囲の子供たちを東に向かわせるが、子供たちは方向を適当に変えてしまう。

このモデルのルールは比較的簡単なので、フローチャートは省略します。

ではモデルの枠組みから作り始めましょう。

1. Universeの下に空間(デフォルト)play groundを作り、teacher(エージェント数20), kid(エージェント数200)を作って下さい。
2. マップ出力の設定もしておきます。
3. モデルにkindergartenという名前を付けて保存して下さい。

次にエージェントのルールです。まず、初期配置として、どちらのタイプのエージェントも空間にランダムに存在し、また向きもランダムとします。Agt_Init{ }のセクションのルール記述は、もうおなじみの設定ですから省略します。

teacherの行動ルールは

```
Agt_Step{
Dim neighbor As Agtset
Dim one As Agt
Turn(Rnd() * 60 - 30)
Forward(1)
MakeOneAgtsetAroundOwn(neighbor, 2, Universe.playground.kid,
False)
For each one in neighbor
  one.Direction = 0
Next one
}
```

他方, kidの行動ルールは次のようにになります。

```
Agt_Step{  
Turn(Rnd() * 60 - 30)  
Forward(0.5)  
}
```

それでは、上書き保存してから、実行して下さい([図17.1](#))。

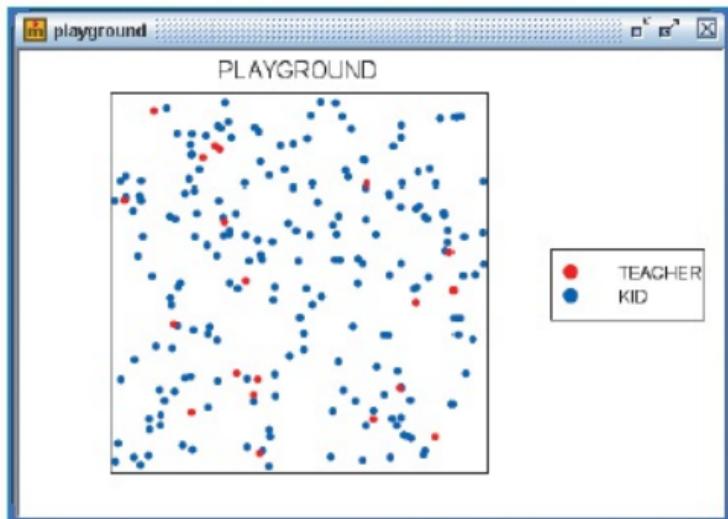


図17.1

以上のルールで、先生が子供たちをなかなか思う方向に向かわせられず、走り回って苦労している様子が観察できると思います。teacherのプロパティ画面を開いて、エージェント数を増やしてみましょう(たとえば50に)。違いに注意してください。

[17.3](#) 状況の把握を容易にする

追加的作業でシミュレーションの実行プロセスを見やすくしましょう。

まず、kidの中で、先生の言いつけをだいたい守っている子供を水色(Color_Cyan)に、そうではない子供を紫色(Color_Magenta)で示しましょう。新しい整数型の変数colorをkidエージェントに追加して、マップ出力も必要な編集(エージェントの色の変数指定)をしておきます。そして、次のようなkidのルールをAgt_Step{ }に追加しましょう。

```
If My.Direction < 30 Or My.Direction > 330 Then  
    My.color = Color_Cyan  
Else  
    My.color = Color_Magenta  
End if
```

次に、Universeのルールを利用して、先生の言うことをきいている子供たちの割合を集計して、時系列グラフに出力しましょう。まず、Universeに整数型変数followを追加します。

UniverseのルールエディタにはUniv_Step_End{ }に次のようなルールを書き込みます。

```
Univ Step_End{  
Dim total As Agtset  
Dim one As Agt  
Universe.follow = 0  
MakeAgtset(total, Universe.playground.kid)  
For each one in total  
    If one.Direction < 30 Or one.Direction > 330 Then  
        Universe.follow = Universe.follow + 1  
    End if  
Next one  
}
```

時系列グラフを出力設定しておきましょう。なお、出力値はUniverse.followそのままではなく、 $100 * \text{Universe.follow} / 200$ としておきましょう。先生の言うことをきいている子供の百分率です。算術としては、言うまでもなく $\text{Universe.follow} / 2$ に等しい値ですが、エージェント数を後で変えたりする場合を考え、わざとわかりやすい計算式にしてあります。

それでは、kindergarten(B)という名前を付けて保存して下さい。実行してみると[図17.2](#)のようになるはず

です。

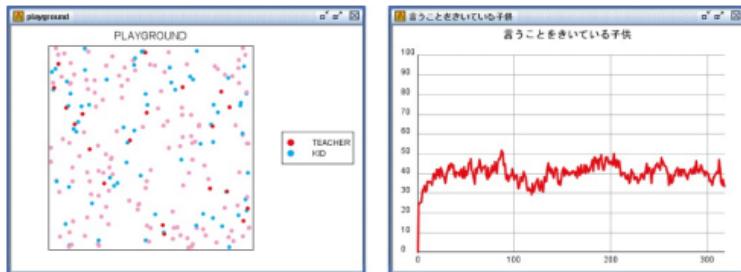


図17.2

17.4 暴走族の取り締まりをモデル化する

次は、禁止(と不服従)の例です。上の幼稚園のモデルより、エージェントの相互作用を複雑に設定してあります。

町中を暴走族が我が物顔にバイクを乗り回しているので、それを警官が取り締まろうとしている。スピード違反のバイクを捕まえて、反則切符を渡す。罰則はだんだん厳しくなるが、暴走族はなかなかかめげない。

具体的には、以下のフローチャートのような設定にします。

警官はパトロールしながら、スピード違反のバイクを捕まえ、違反者に応じた反則切符を手交します。

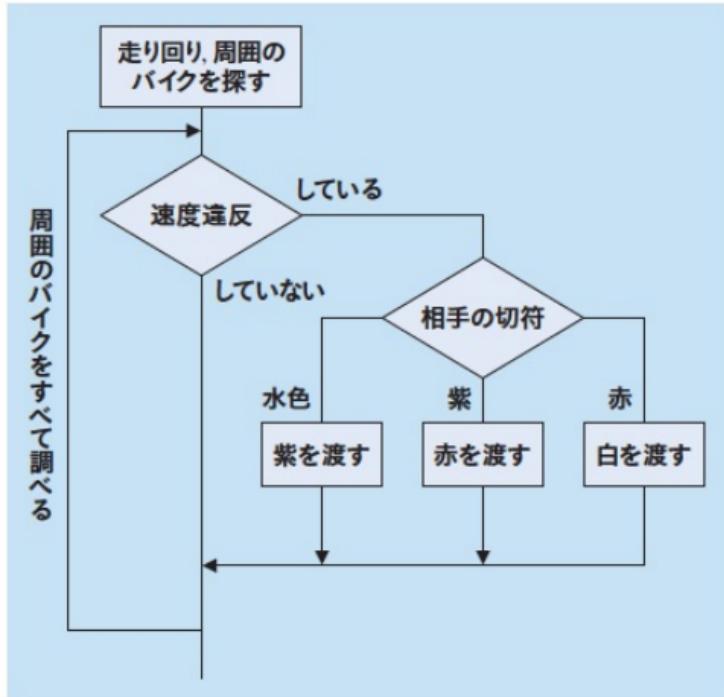


図17.3

暴走族は、反則切符の厳しさに応じて、バイクのスピードを落としますが、素直に制限速度以下にはしません。

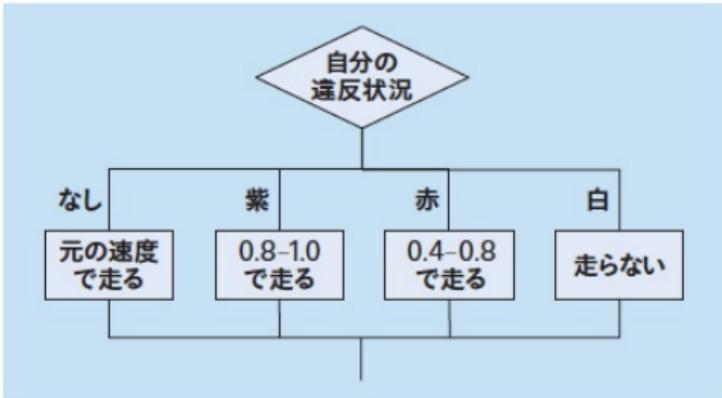


図17.4

では準備作業です。

1. streetsという空間(デフォルト)を作り, biker(エージェント数100)とpolice(エージェント数20)の2種類のエージェントを作ります。
2. bikerには反則状況を表すための整数型変数fineとスピードを表す実数型変数speedを追加します。
3. マップ出力を設定します。なお, bikerの色表示はfineによる変数指定とします。
4. モデルをcrackdownという名前で保存しましょう。

ルールの書き込みに移ります。Universeにはルールを書き込みません。警官と暴走族の初期配置は、いつものように、空間全体にランダムに配置して下さい。動く方向もランダムです。暴走族の動く速さは1(各ステップ1動くことが時速100キロに相当することにしましょう)です。反則の段階は、水色(ゼロ), 紫色(1回), 赤色(2回), 白色(3回)とします。最後を白にしたのは、3度目の違反でバイクを乗り回せなくなったエージェントをマップ上で見えにくくするためです。初期状態は、全員反則ゼロ(水色)にします。初期状態(Agt_Init{})のルールの記述は省略します。

警官の行動ルールは、次のようにします。自分の周りの暴走族の運転状況を観察して、適切な取り締まりをする必要があります。この町での制限速度は時速60キロ(つまり0.6)と仮定しましょう。

```
Agt_Step{
Dim gang As Agtset
Dim kid As Agt
Forward(0.1)
Turn(Rnd() * 60 - 30)
MakeOneAgtsetAroundOwn(gang, 0.5, Universe.streets.biker, False)
For each kid in gang
  If kid.speed > 0.6 Then
    kid.speed = 0
    If kid.fine == Color_Cyan Then
      kid.fine = Color_Magenta
    Elseif kid.fine == Color_Magenta Then
      kid.fine = Color_Red
    Elseif kid.fine == Color_Red Then
      kid.fine = Color_White
    End if
  End if
Next kid
}
```

上のルールでは、警官の暴走族に反則切符をきる行為は、他種エージェントの属性を変えるというartisocの技法になっています。

他方、暴走族bikerのルールは次のようにになります。

```
Agt Step{
If My.speed == 0 Then
  If My.fine == Color_Magenta Then
    My.speed = 1 - Rnd() * 0.2
  Elseif My.fine == Color_Red Then
    My.speed = 0.8 - Rnd() * 0.4
  End if
End if
Forward(My.speed)
}
```

暴走族については、自分の属性の変化に応じて自分の行動を変えるという、マルチエージェント・シミュレーションのオーソドックスな技法を使っています。

このように行動ルールは単純です。レパートリーは4つです。どの場合になるのかが、警官による取り締まりで決まるのです。もっとも違反を3回してしまった場合以外、反則の履歴があるからといって、必ずしも、スピード違反しないように気をつけるわけではありません。

上書き保存してから、実行して下さい(図17.5)。

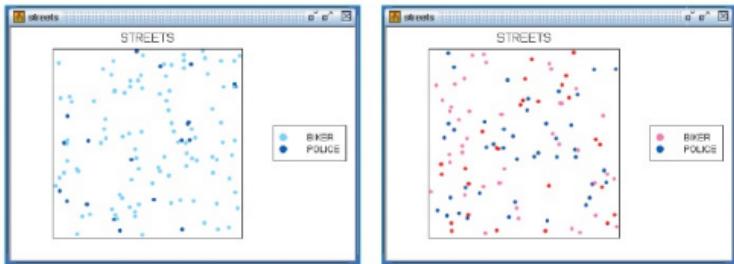


図17.5

17.5 取り締まりの効果を調べる(とばしてもかまいません。)

暴走族の取り締まり状況を、リアルタイムで見てみましょう。平均時速とバイクを乗り回している暴走族の人数を表す実数型変数avspeedとactiveをUniverseの下に追加します。両方をそのままの値で時系列グラフで出力するように設定して下さい。バイクを乗り回している暴走族がいなくなった段階で、シミュレーションを終了させるようにしましょう。Universeには次のようなルールを書き込んで、集計作業を行ないます。

```
Univ_Step_End{
Dim gang As Agtset
Dim kid As Agt
Universe.avspeed = 0
Universe.active = 0
MakeAgtset(gang, Universe.streets.biker)
For each kid in gang
  If kid.fine == Color_White Then
    Else
```

```
    Universe.avspeed = Universe.avspeed + kid.speed
    Universe.active = Universe.active + 1
End if
Next kid
If Universe.active == 0 Then
    ExitSimulationMsgLn("No More Active Bikers")
Else
    Universe.avspeed = 100 * Universe.avspeed / Universe.active
End if
}
```

上のルールで、平均時速を求める計算をどこでしているのかに注目して下さい。最後のイフ文の中です。これは、Universe.activeがゼロの時にわり算をしないようにするためです。

この集計作業を加えたモデルは、crackdown(B)という名前を付けて保存してください。

17.6 命令よりは同調が重要？

最後に、他のエージェントに働きかける問題と他のエージェントから影響を受ける問題とを同時に扱ってみます。この章の初めに作ったkindergarten(B)モデルを修正しましょう。

幼稚園のモデルでは、子供たちは周囲の子供の状況におかまいなく、勝手にうろうろする行動ルールでした。それでは、大人ほどではないにしても、子供たちも周囲から少しは影響を受けるとするとどうなるでしょうか？

そこで、第13章で学んだ、他のエージェントから影響を受ける技法をそのまま利用して、kidのルールエディタの中に次のような同調するルールを加えてみましょう。

周囲の子供たちのほとんどが言うことを聴いているとようやく自分も言うことを聞く。

幼稚園のモデルkindergarten(B)を開いてから、別の名前、たとえばkindergarten(C) という名前を付けて保存します。teacherの行動ルールはそのままにしてあります。kidの行動ルールについて、17.2で書き込んだルールの代わりに、次のような新しいルールを記述します。なお、17.3で書き込んだ色付けルールは

残しておきます。

1. 周囲の仲間を調べる(視野の広さ!)
2. 周囲に仲間がいて、そのほとんど(たとえば8割)が東の方を向いていれば、自分もそうする

このルールもMakeOneAgtsetAroundOwn()とフォーマット文を使って記述できます。

全体のルールは次のようになります。

```
Agt_Step{
Dim neighbor As Agtset
Dim one As Agt
Dim num As Double //後でわり算するので実数型
Dim count As Double //後でわり算するので実数型
MakeOneAgtsetAroundOwn(neighbor, 1, Universe.playground.kid,
False)
num = CountAgtset(neighbor)
If num == 0 Then
    Turn(Rnd() * 60 - 30)
Else //周囲に子供があれば
    count = 0 //念のために初期化する
    For each one in neighbor //周囲の子供の状態を調べる
        If one.color == Color_Cyan Then
            count = count + 1
        End if
    Next one
    If count / num >= 0.8 Then //従順な子供が多ければ
        My.Direction = 0
        Turn(Rnd() * 30 - 15)
    Else //さもなければ
        Turn(Rnd() * 60 - 30)
    End if
End if
Forward(0.5)
//(以下、17.3で追加したルールが続く)
}
```

上書き保存してから、実行ボタンを押してみましょう。周囲から影響を受けるというルールに変えると、状況が激変するのが観察されます。

この事例は、多数の人に同じ事をさせるのに、命令よりも同調がいかに重要か、を示唆しています。シ

ンブルですが、とても興味深いモデルになっています。

新しく学んだ事項

- 文法的には新しく登場したものはありません
- 他のエージェントの属性の複雑な変え方

練習問題

17.1

kindergarten(C)モデルをさらに改良して、パラメータを変えることによりシミュレーション結果にどのような違いが現れるか、観察してみよう。

先生の人数や、子供の人数をコントロールパネルで変えられるようにする。



Universeに適当な変数を追加し、Univ_Init{ }に適当なルールを書き込み、Agt_Init{ }はどちらのエージェント種からも不要になったルールを削除します。プロパティ・ウインドウでエージェント数をゼロにすることを忘れずに。

子供の数よりは先生の数が重要なことがわかる。

17.2

練習問題17.1の続きである。

実は、先生の走り回る能力Forward()ないし先生の人口密度、あるいは周囲の子供たちへの影響範囲(視野の広さ)が、子供を一定方向に歩かせるのに重要なファクターかも知れない。ルールエディタの中を適当に書き換えて、調べてみよう。

いずれにせよ、子供たちに言うことを聽かせるのは大変です。

17.3

取り締まりのモデル(crackdown.model)で、暴走族の行動を変えてみよう。

取り締まられたライダーはいったん停止するが、周囲に警官が1人しかいないと、反則切符を受け取らずに逃走する。



周囲を調べるルールをどこに書けばよいでしょうか。

第18章

空間を「場」として利用する

18.0 空間には意味があります

- 空間に役割や機能を持たせることができます
- 空間変数という新しいタイプの変数を設定します
- 空間上の「立ち位置」でエージェントが影響を受けます
- 「場」や「勾配」をマップ出力で色の違いで可視化できます
- 空間変数の値は、シミュレーションの過程で変化可能です
- 深海魚のモデルで小手調べ
- 空気感染をモデル化しましょう

18.1 「場」や「勾配」は難しくない

これまででは、エージェントは空間の中で行動してきました。ただ、空間は単にエージェントがどこに位置するのかを表す2次元の座標系としてしか使われてきませんでした。このことは、X軸を人口、Y軸を経済規模など、物理的な空間と見なさない場合でも同じでした。それに対して、この章では、空間にもっと重要な役割ないし機能を持たせて、エージェントの行動に影響を及ぼす技法を学びます。

たとえば、電磁場の中では電荷を持った粒子の動きに変化が生じます。肥沃な農地では作物が良く育ち、瘦せた農地では貧弱な野菜しか採れません。工場は地価の高い都心からやすい郊外に移転します。明るい方をめざして泳ぐプランクトンもいれば、フェロモンの濃い方に集まる虫もいます。このように、空間は一般的に「場」と呼ぶ性質を持つことがあります。「場」では、その地点の値だけでなく、近くの地点との値の差（勾配）が大きな意味を持つこともあります。登山をしていると、勾配のきつい山道は登りも下りも大変です。

このような「場」や「勾配」の情報は、2次元空間(曲面、平面)上に表すことが可能で。たとえば地形図では、紙(平面)の上に高度(海拔)が等高線を引くという技法で示されています。これにより、高さだけでなく、勾配(等高線の間隔の粗密)も分かります。artisocの空間も、このような「場」にすることが可能で。

本格的な空間の利用、特に「勾配」をルールに組み込んだモデルは第4部(第34章)で学ぶことにして、この章では空間の利用法の基本を学びます。

18.2 空間変数を利用する

まず、空間を「場」として利用することに慣れましょう。その基本は、空間に変数を設定し、その変数値を色でマップ上に表すことです。設定された空間変数の値は、エージェントの行動ルールに影響を与えることができます。

海がだんだん深くなる。

Universeの下に空間(デフォルト)seaを作ります。今まででは空間の下にエージェントを作りましたが、ここではdepthという整数型変数を追加します。このように、空間に直接追加する変数のことを「空間変数」と呼んで、特別の扱いをします。何が特別かというと、空間変数は必ず空間座標の上に定義されているのです。つまり、空間変数はUniverse.sea.depth(i, j, 0)のように記述します。最後の(i, j, 0)が、エージェントと同じような空間上の座標で、iはX座標、jはY座標を表します。最後の0は、エージェントの変数でいうレイヤ(Layer)ですが、まだ学んでいないので、とりあえず0を予め書き込んでおきます。

ただし、エージェントの座標が実数型だったのに対し、空間変数の座標(i, j, 0)のi, jは整数値しかとりません。空間変数は、空間座標の整数値に対して設定される点に注意して下さい。本来なら、場や勾配を表す変数は連続的な空間上に連続的な値をとるはずですが、artisocではあたかも空間が格子状になっているものとみなして、空間変数がとることのできる座標の値を整数にしているのです。イメージとしては、格子を設定したときの墨線の交点が空間座標の各点に対応しています。したがって、空間変数を細

かく設定したければ、空間を大きくして下さい。

さて、Universeのルールエディタには、次のようなルールを書き込みます。

```
Univ_Init{
Dim i As Integer
Dim j As Integer
For i = 0 To 49
    For j = 0 To 49
        Universe.sea.depth(i, j, 0) = i + j
    Next j
Next i
}
```

50×50の空間では、X座標もY座標も整数值では0から49までの値になります。つまり、Universe.sea.depth(i, j, 0)のiとjの範囲は0から49までになります。本文が「入れ子」構造になっている点に注意して下さい。上のルールだと、Universe.depthの値は、原点(左下、i = 0, j = 0)では0になり、そこから右上(i = 49, j = 49)の98までなだらかに変化します。

海の深さ(とその変化)は、マップ出力することができます。つまり、空間変数は、エージェント種と同じように、マップの上に表すことができるのです。ただし、設定方法が少し異なります。まず、普通にマップ出力設定画面で設定します。マップ要素を追加するための選択メニューに空間変数depthがあるはずなので、depthを選択します。すると、エージェント種を追加するときとは異なり、[図18.1](#)のような設定画面が現れます。色選択ボタンを押すと、色設定用のパレットが出てきますから、それぞれ薄い青と濃い青を選択しましょう([図18.2](#))。それを図のように0から100(98でもよいのですが、とりあえず100にします)まで変化するように設定します。この際、マーカー■を選択すると、背景のように見えるのできれいです。

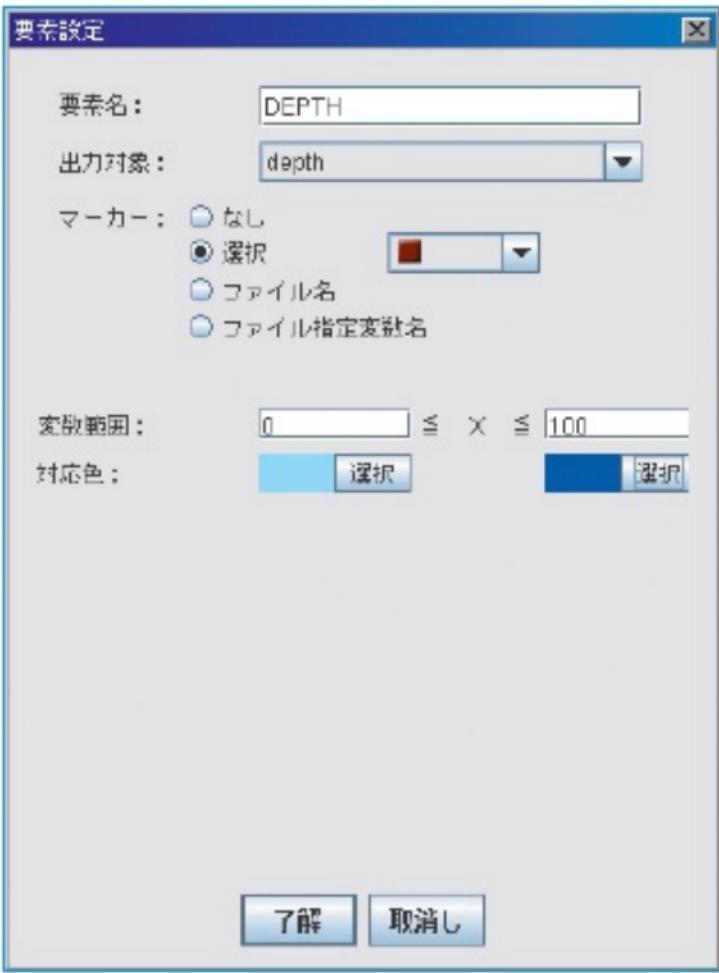




図18.2

ここまでのお作業をしたら、oceanという名前を付けて保存して下さい。そして、ステップ実行ボタンを押してみましょう。[図18.3](#)のようなマップが現れます。

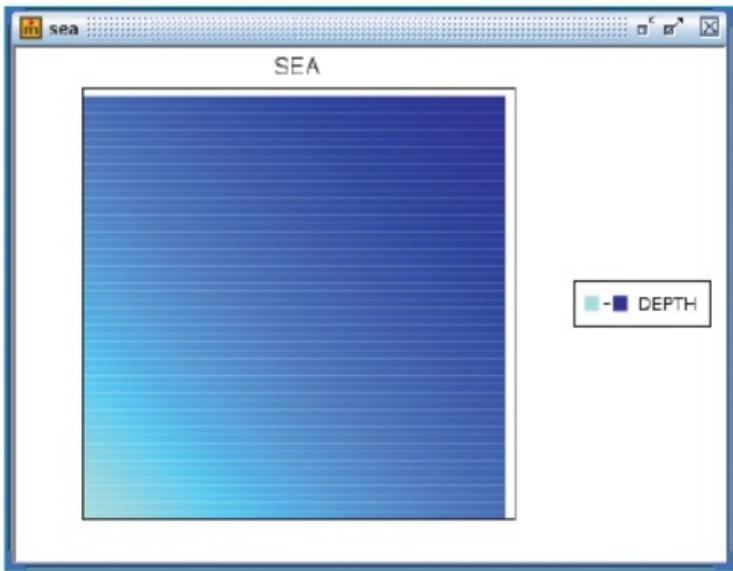


図18.3

このように、空間を場として利用するには、「空間変数」というものを利用します。空間変数の値を出力マップに反映させて、空間に色づけすることで、「場」や「勾配」を直感的に把握できるのです。

18.3 深海魚の泳ぐ海

`ocean`モデルの海に、魚を泳がせてみましょう。

しかしその前に、海の深さの設定にちょっとした工夫が必要です。[図18.3](#)のマップを見ると、空間はループしているので、海の最も深い所(右上)と最も浅い所(左下)とがすぐ近くに位置しています。これは不自然です。ループのある空間では、上下の縁と左右の縁との値を近づけて、勾配を全体的になだらかになるようにしておく必要があります。

海の深さを中央部分が最も深く、周辺に行くにしたがって浅くなるように設定する。

こうすることによって、ループしていても、隣接している上端と下端、右端と左端は浅瀬になり、滑らかになるはずです。

ルールは、やや技巧的ですが、次のように修正すると簡単です。

```
Univ Init{
Dim i As Integer
Dim j As Integer
Dim k As Integer
Dim l As Integer
For i = 0 To 49
    For j = 0 To 49
        k = Abs(i - 24) * 2
        l = Abs(j - 24) * 2
        Universe.sea.depth(i, j, 0) = 100 - k - l
    Next j
Next i
}
```

ここで $k = \text{Abs}(i - 24) * 2$ という表記が出てきました。この中の $\text{Abs}(i - 24)$ とは $(i - 24)$ の絶対値をとるという意味です。要するに、 i が 0 から 24 に増えると、 $\text{Abs}(i - 24)$ は 24 から 0 に減るので、結局は 48 から 0 に変化します。さらに、 i が 25 から 49 に変化すると、 k は 2 から 50 に変化します。これで、 k は i が 24 のときに最小になります。 l についても同様です。

したがって、 Universe.depth は (24, 24) で最大の 100 となり、(0, 0) で 4、(49, 49) で 0 となります。こうして、海の中心部分が最も深くなり、周辺ほど浅くなって、上下左右の縁では、ループしても海の深さはほぼならかにつながります。

このような技巧の理屈さえ理解できれば、 k や l をわざわざ一時的変数として宣言せず、 $\text{Universe.sea.depth}(i, j, 0) = 100 - \text{Abs}(i - 24) * 2 - \text{Abs}(j - 24) * 2$ と表記しても差し支えありません。

それでは、ocean(B)という名前を付けて保存して、実行ボタンを押してみましょう。海の深さの状態が大きく変わったことと思います(図18.4)。

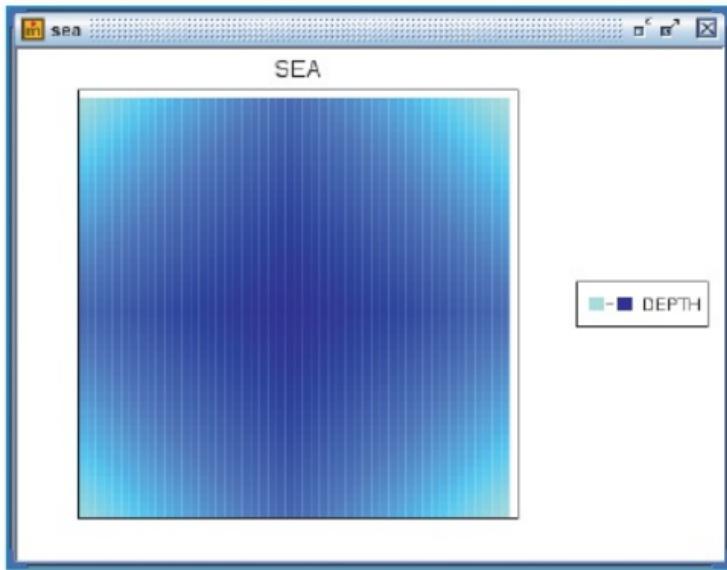


図18.4

18.4 深海魚の行動

では魚を泳がせる準備に入りましょう。seaにfishというエージェント(数は10)を追加します。マップ出力の編集により、fishの要素設定もしておいてください。マーカーを▲にすると、泳いでいる雰囲気が出ますよ。なお、fishの要素設定を終えたら、マップ要素リストに注目して下さい。設定した順に、上からdepth、fishと並んでいませんか(図18.5左図参照)。このままだと、空間変数depthが最前面になり、fishのアイコンは後ろに隠れて見えなくなります。そこで、マップ要素リスト・ウインドウの右にあるボタンを適当に操作してマップ要素リストでfishを上にdepthを下に置くようにして下さい(図18.5右図参照)。



図18.5

深いところで生活する魚(深海魚)は、浅い方へ泳いでいくと反転する。

ここでは、場(海の深さ)の魚への影響に注目するために、魚どうしの相互作用は何も考えません。初期設定も単純にしましょう。最初は、最も深い中心部にいるようにします。空間の中央にエージェントを配置する関数MoveToCenter()を使いましょう。魚のいる海の深さは、魚の近くの空間座標で近似しましょう。魚(エージェント)の位置座標は実数型ですが、空間変数の位置座標は整数型です。そこで、魚の位置座標の小数点以下を切り捨てて、近くの空間座標の座標値としましょう。

```

Agt_Init{
MoveToCenter()
My.Direction = Rnd() * 360
}
Agt_Step{
Dim xi As Integer
Dim yi As Integer
xi = Int(My.X)
yi = Int(My.Y)
If Universe.sea.depth(xi, yi, 0) <= 60 Then
    Turn(180)
Else
    Turn(Rnd() * 40 - 20)
End if
Forward(0.5)
}

```

ocean(C)という名前をつけて保存してから、実行ボタンを押して下さい。魚は、海の中心部分で泳ぎ

回っているでしょうか(図18.6)。

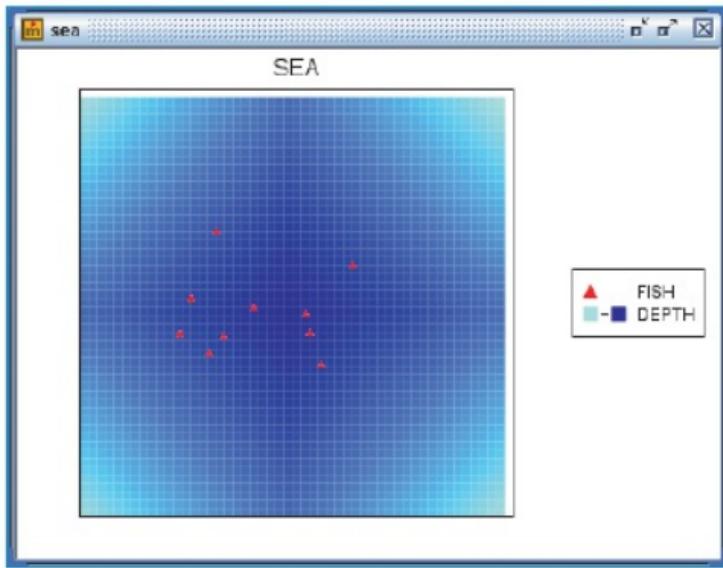


図18.6

18.5 空気感染をモデル化する

空間変数を媒介にして、エージェント間の本格的な相互作用をモデル化しましょう。上で作ったoceanモデルでは、海の深さを表す空間変数は最初に設定され、シミュレーションの過程では変わりませんでした。ここでは、シミュレーションの中で空間変数の値が変化するモデルを作ります。

第13章で、感染症の流行モデルを作りました。そのとき、健康な人はそばに患者がいると病気になるというルールでした。しかし実際には、患者と健康な人を関係づけているのは患者から出る病原体(細菌やウィルス)です。病原体を別種のエージェントとしてモデル化することもできますが、ここでは、空間変数を利用しましょう。

患者は自分の近くに病原体を残す。健康人は自分の近くに病原体が多いと病気になる。病原体は徐々に減っていく。

第13章で作ったinfection(D)モデルは、動き回る人々がいる中で、病人がそばにいると自分も病気になり、病人はやがて回復するモデルです。それを修正して、新しいモデルを作ります。いわば、接触感染の流行モデルから空気感染の流行モデルへと作り替えるのです。

モデルを作る手間を省くために、infection(D)を利用してしまおう。infection(D)を開いて、それにinfluenzaという別の名前を付けて保存して下さい。これから、この新しく保存したモデルに手を加えます。

まず、society空間に整数型変数virusを追加します。この空間変数で、各位置における病原体の数を表すものとしましょう。influenzaモデルに既に書き込まれているルールをベースにして、新しいルールを書き込んだり、差し替えたりします。

1. Universeとpersonについての初期設定

Univ_Init{}とAgt_Init{}は、そのまま利用します。

2. 患者は病原体を空間に残し、健康人は近くの病原体の数によって病気になったりならなかったりする。

具体的には、患者は毎ステップ、自分の近くの空間に病原体を3個放出するルールにしましょう。ここで自分の近くの空間とは、自分の位置座標(X, Y)の値を小数点以下を切り捨てた値の座標をもつ空間としましょう。そして、健康人は近くの空間の病原体の数が2個以下なら感染しないが、それを越えると多いほど感染しやすいというルールにしてみましょう。Agt_Step{}を下のルールに差し替えることにします。

```
Agt_Step{
    Dim xi As Integer
    Dim yi As Integer
    xi = Int(My.X)
```

```

yi = Int(My.Y)

If My.condition == Color_Red Then
    Universe.society.virus(xi, yi, 0) =
    Universe.society.virus(xi, yi, 0) + 3
    my.cure = my.cure + 1
Else
    If Universe.society.virus(xi, yi, 0) * Rnd() > 2 Then
        My.condition = Color_Red
    End if
End if

Turn(Rnd() * 20 - 10) //以下, infection(D)モデルと同じ
Forward(1)

If My.cure >= 7 Then
    My.condition = Color_Blue
    My.cure = 0
End if
}

```

infection(D)モデルでは自分の周囲にいる患者の数に応じて感染したりしなかったりするルールでしたが、ここでは、近くの病原体の数(virusの値)に依存するようになっています。周囲を認識させるルールが不要になったので、かえってすっきりしました。

3. 空間上の病原体が徐々に減っていく。

Univ_Step_End{ }のセクションの最初に次のようなルールを追加します。毎ステップ、virusが1個ずつ消えていくルールです。なお、下記ルールの薄字の部分は、既に書き込まれているはずのinfection(D)のルールです。

```
Univ_Step_End{
```

```

Dim people As Agtset
Dim one As Agt
Dim i As Integer
Dim j As Integer
For i = 0 To 49
    For j = 0 To 49
        If Universe.society.virus(i, j, 0) > 0 Then
            Universe.society.virus(i, j, 0) =
Universe.society.virus(i, j, 0) - 1
        End if
    Next j
Next i
Universe.number = 0
MakeAgtsetSpace(people, Universe.society)
For each one in people
    If one.condition == Color_Red Then
        Universe.number = Universe.number + 1
    End if
Next one
}

```

ルール変更は以上で完了です。

マップ出力にvirusをマップ要素として追加して下さい。空間変数の値に応じて、マップ上の色を変えるわけですが、色の出し方については、さまざまな工夫が可能です。一般論として、要素設定ウィンドウで選択する、最小値と最大値の間の数と任意の2色の間のグラデーションとが対応します。したがって、最小値と最大値は、ルールの中で空間変数にどのような値を与えるのかを反映させる必要があります。influenzaモデルでは、病人が出すvirusの数、virus数に応じて罹患する確率、virusが死滅していく率な

どを勘案して、たとえば最小値0で白色、最大値5で紫色を選ぶと、personの色とも区別できて、見栄えが良くなります(図18.7参照)。



図 18.7

上書き保存してから実行ボタンを押しましょう。

空間変数はいろいろな場面で利用できます。技巧的な利用例を含めて、第4部(第34章)で詳しく取り上げます。

新しく学んだ事項

- 直接、空間に変数を追加する(空間変数の追加)
- 空間変数は空間に格子状に設定されている(整数座標しかとれない)
ループしている空間の勾配をなめらかに設定する技巧的な技法
- シミュレーション実行中に空間変数の値を変える
- 空間変数の値を色でマップ出力する
- マップ要素リストの順番の意味と順番の変更法
- 空間変数のいくつかの利用方法
- フォ文の「入れ子」構造
- MoveToCenter()

- Abs()

練習問題

18.1

18.3で作ったocean(B)モデルに、浅い海だけを泳ぎ回るサメを何匹か追加する。



If Universe.sea.depth(Int(My.X), Int(My.Y), 0) >= 60 Then

18.2

患者が毎ステップ放出する病原体の数をコントロールパネルで1から6まで調節できるようにinfluenzaモデルを修正する。



エージェントのルールの変更を忘れずに。

初期の感染率(既にinfection(B)でコントロールパネルを作ったので残っているはずです)や、放出病原体の数をいろいろと変えることにより、流行現象の違いを観察しましょう。

第19章

同期問題に注意する

19.0 シミュレーションの中の時刻と時間を意識しましょう

- シミュレーションの中の時刻と時間に注意する必要のある場合があります
- 注意するポイントは「同期」という問題です
- 沢山いるエージェントの「実行順序」の問題でもあります
- 過去を現在に(現在を未来に)繋げるエージェントの「記憶」の問題でもあります
- 簡単なモデルで、問題の所在をはっきりさせます
- 問題を解決するには2つのやり方があります

19.1 シミュレーションの中の時刻と時間

第11章から第17章にかけてエージェントどうしに相互作用をさせる技法を学びました。これはマルチエージェント・シミュレーション技法の中核です。この技法を使うと、エージェントたちは互いに自分自身の行動ルールにしたがって「勝手に」どんどん相互作用をします。各ステップで生じている相互作用は、ルール次第で複雑になります。つまり、1ステップの間にモデルの中ではさまざまな現象が生じているのです。

そこで、たくさんのエージェントが「同時」に行動するということが、実際にはどのような現象になっているのか理解する必要があります。それと同時に、実際に生じている現象を、本当に「同時」に起こっているようにルールを書き込む必要性も出できます。これをマルチエージェント・シミュレーションの「同期(シンクロナイゼーション)」問題と言います。エージェントを「シンクロ」させる、などと使うこともあります。シンクロナイズド・スイミングの「シンクロ」と同じ意味です。

この章では、同期をめぐる問題の基本を扱います。特に、エージェントの行動を「同期」させる方法の基本を学びます。同期問題を含む一般的な実行順序の問題は第4部(第31章)で扱います。

19.2 なぜ「同期」が重要なのか

むずかしい理屈はさておき、具体的な例を見てみましょう。まず、次のようなモデルを作って下さい。

山に木が生えている。山火事が起こると、燃えている木に隣接する木にも火が移り、木が次々と燃えていく。

Universeの下にwoodsという空間を作ります。ただし、デフォルトではなく、空間の大きさを 8×2 にして、「ループしない」を選択して下さい。woodsの下にtreeエージェント種を作ります。エージェントはUniverseで作るので、エージェント数は0(デフォルト値)にしておいて下さい。そしてtreeには燃えているかどうかを表す変数colorを整数型で作ります。マップ出力を設定しますが、treeをマップ要素に追加するとき、色を変数指定(color)してください。forest-fireという名前で保存して下さい。

まず、木(treeエージェント)を林(woods)に順序よく植えます。左下の木だけが最初燃えているようにします。

```
Univ_Init{
Dim one As Agt
Dim i As Integer
Dim j As Integer
For i = 0 To 7
  For j = 0 To 1
    one = CreateAgt(Universe.woods.tree)
    one.X = i
    one.Y = j
    If i == 0 And j == 0 Then //左下隅なら
      one.color = Color_Red
    Else //左下隅以外なら
      one.color = Color_Green
    End if
  Next j
Next i
}
```

上のルールで、 8×2 の空間に木がぎっしりと生えており、左下の木が1本だけ燃えている状態になっています。上書き保存してから、実行ボタンを押して確認して下さい。

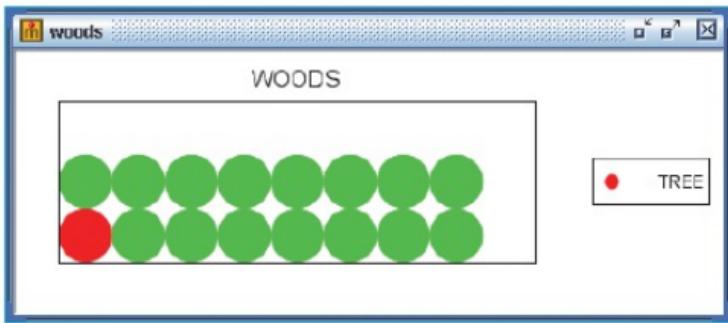


図19.1

隣接している木が燃えていれば火が燃え移り、燃えている木は燃え尽きるルールにしましょう。treeエージェントの行動ルール(つまり、燃えるかどうか)は[図19.2](#)のようなフローチャートになります。

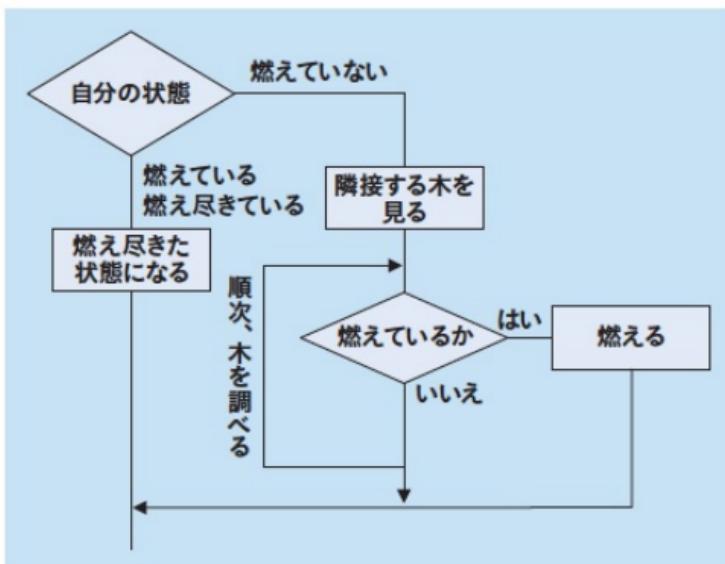


図19.2

これをルール化すると次のようになります。

```
Agt_Step{
Dim neighbor As Agtset
Dim one As Agt

If My.color == Color_Green Then
    MakeAllAgtsetAroundOwn(neighbor, 1, False)
    For each one in neighbor
        If one.color == Color_Red Then
            My.color = Color_Red
            Break //もう調べる必要がない
        End if
    Next one
    //Breakが実行されると、ここ(フォ文の外)にでる
Else //今燃えている木が何本あったとしても、もう燃え尽きていても結局同じ
    My.color = Color_Black
End if
}
```

上のルールで、新しい表現Breakが登場しました。Breakが使われているのは、フォ・イーチ文の中のイフ文の中です。Breakとは、ただちに繰り返し文による繰り返し作業を中止するルールです。イフ文の条件がいったん満たされてルールが実行されれば、さらに繰り返す必要がないときに実行させるルールです。上の場合、周囲に燃えている木が何本あったとしても、フォ・イーチ文で順番に調べていき、燃えている木が見つかれば、もうそれ以上調べる必要がありません。なお、Breakはフォ文でも使えます。

ここで、forest-fire(B)という名前を付けて保存しましょう。本来なら、[図19.3](#)のように延焼するはずです。

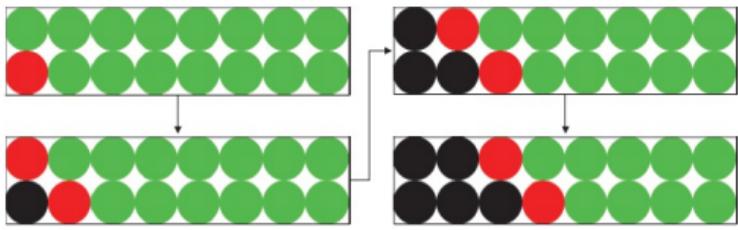


図19.3

これからが重要です。ステップ実行ボタンを1回ずつ、マップ出力の変化に注意しながら押して下さい。変化がなくなったら停止ボタンを押して、いったんシミュレーションを終了し、再びステップ実行ボタンを押すことを何回も繰り返して下さい。実際には、[図19.4](#)のようなさまざまな結果になったはずです。

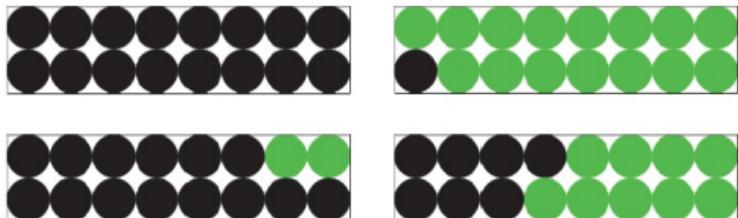


図19.4

このように、実際のシミュレーション結果は、「燃えている木に隣接する木に火が移り、次々と木が燃えていく」と想定したシミュレーションになっていません。なぜ、こうなるのか考えてみましょう。

[19.3](#) artisocは何をやっているのか

初期状態に戻って考えてみます。

説明の便宜のために、木に番号を与えておきます。[図19.5](#)で、燃えている木(0)に隣接しているのは1と8だけです。したがって、「次の瞬間」に燃えているのはこの2本の木だけ(赤色)のはずです。そして、初期に

燃えていた木0は「次の瞬間」には燃え尽きている(黒色)はずです。

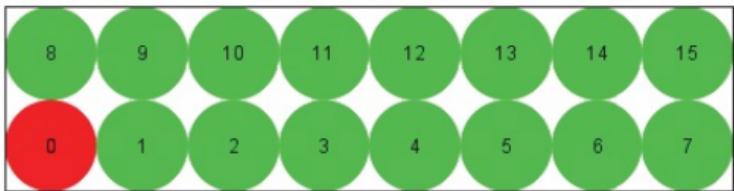


図19.5

たしかに、全ての木が「同時」に「次の瞬間」に火が移ってくるかどうか判断できたら、このような結果になるでしょう。しかし現実には、artisocは各treeエージェントに同時並行的にルールを実行させることはできません。artisocは各エージェントのルールを順次実行していきます。したがってartisocのなかの「同時」とは、artisocが最初のエージェントのルールを実行し始めてから最後のエージェントのルールを実行し終わるまでの「長い時間」なのです。もっと正確にいうと、Univ_Step_Begin{}からUniv_Step_End{}までです([コラム「Universeのルールエディタが複雑なのはなぜ?」](#)を参照)。

artisocはルールを実行するエージェントの順番を、毎ステップ、ランダムに決めています(設定を変えることもできますが、これは第4部(第31章)で学びます)。もし上のような初期状態が与えられたとき、次の「瞬間」にartisocが燃えている木(0)のルールを最初に実行すると、行動ルールにしたがって、燃え尽きてしまいます。隣接する1番の木のルールが次に実行される順だとしても、燃えている木はありません。したがって、延焼は全然起りません。では初期状態の次の「瞬間」に1, 2, 3, 4, 5, 6.....15, 0という順序でエージェントのルールが実行されるとしましょう。すると、いつも隣接する木が燃えている状態ですから、「一瞬」(1ステップ)のうちに全ての木が燃え尽きてします。

つまり、私たちが「同時」だと思っている「一瞬」は、実はマルチエージェント・シミュレーションを実行しているartisocにとっては「長い時間」であり、その間にいろいろなことが實際には起こっているのです。そこで、コンピュータが「長い時間」をかけてやっていることを私たちが「同時」とか「一瞬」だと思っていることに合わせる必要がある場合も出てきます。シミュレーションにおける「同時」的現象(具体的には、artisocが1ステッ

のうちに実行しているさまざまのこと)を私たちが直感的に理解している「同時」的現象のように工夫することを「同期」させるというのです。

artisocでは、同期させる方法は、基本的に2種類あります。上の山火事の例でいえば、状態の変化(燃え始める、燃え尽きる)をただちに実行しないで、各ステップの最後に一括して行なう方法と、周囲の木の現在(そのステップ)の状態ではなく1期前の状態に応じて、直ちに状態を変化させる方法です。以下で、この両方を説明します。

19.4 最後にまとめて状態を変える

そこで、現在の木の状態を表すcolorの他に、「次の瞬間」にどのような状態になるのかを示す別の整数型変数condを追加しましょう。そして、仮に状態が変わる(燃え始める、燃え尽きる)にしても、ただちにcolorを変えるのではなく、condを変えておきます。つまり、ルールに「ワン・クッション」を置くのです。薄字のルールは、既に記入済みのものです。

```
Agt_Step{
Dim neighbor As Agtset
Dim one As Agt

If My.color == Color_Green Then
    My.cond = Color_Green
    MakeAllAgtsetAroundOwn(neighbor, 1, False)
    For each one in neighbor
        If one.color == Color_Red Then
            My.cond = Color_Red
            Break //もう調べる必要がない
        End if
    Next one
    //Breakが実行されると、ここ(フォ文の外)にでる
Else //今燃えても、もう燃え尽きていても結局同じ
    My.cond = Color_Black
End if
}
```

そして全エージェントのルールが実行されたあと、Universeのルールとして、一括して置き換えます。

```
Univ_Step_End{
Dim total As Agtset
```

```
Dim one As Agt
MakeAgtset(total, Universe.woods.tree)
For each one in total
    one.color = one.cond
Next one
}
```

これをforest-fire(C)という名前で保存して、またステップ実行ボタンを1回ずつ押してみましょう。今度は、予想したとおりに延焼していくはずです。

つまり、同期するひとつの方法は

1. エージェント種に次の瞬間に変わる状態を指定しておく変数(これを「バックアップ変数」と呼びます)を追加し、行動ルールとしてはその変数を変化させて、現在の状態を表す変数は変化させない。
2. 最後に、一括してバックアップ変数を、状態を表す変数に代入する。

これは、「状態変化先延ばし」法とか「一括置換」法とでも呼べるでしょう。

19.5 過去の状態を参照する

もうひとつの方法は、自分自身が「一瞬前」に燃えていれば「今」燃え尽きるし、隣接している木が「一瞬前」に燃えていれば自分は「今」燃える、という考え方方に基づいています。つまり、木は、「現在」の周囲と自分との関係ではなく、「一瞬前」の周囲や「一瞬前」の自分の状態を基にして「現在」の自分の状態が決まる、という方法です。

この方法をルールとして記述するには、「1ステップ前の過去」の状態を知る必要があります。artisocには、各変数の過去の値を保存したり、過去の値を呼び出したりすることのできる機能が備わっているので、それを利用することにしましょう。それは、

1. 変数のプロパティを開いて、その変数が自身の過去の値を記憶しておける状態にする
2. GetHistory()というルールを用いて、過去の状態を認識する

という方法です。

もう一度、forest-fire(B)モデルを開いて下さい。まず、colorのプロパティを開き、記憶数(デフォルトは0)を1に設定して、過去1期前の状態を記憶させておくようにします(図19.6参照)。forest-fire(D)という名前で保存しましょう。

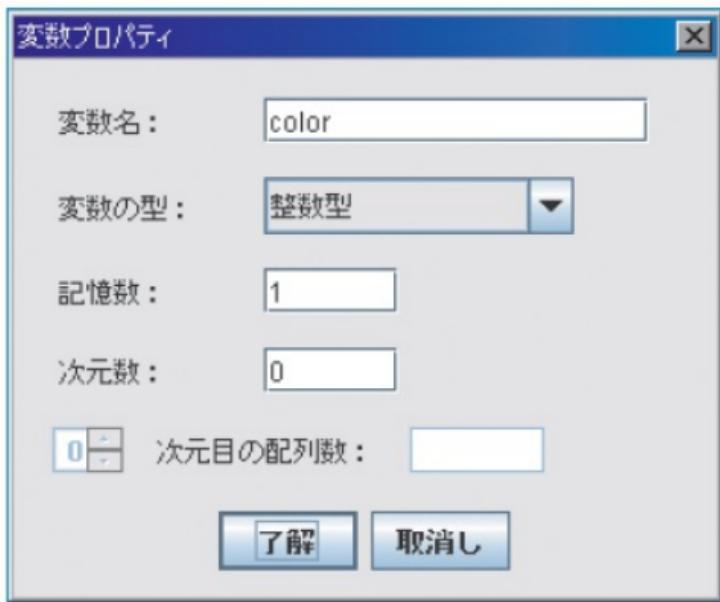


図19.6

その上で、たとえばGetHistory(My.color, 1)と書き込むと、自分の状態(color)の1ステップ前の最終状態を認識させることができます。

この方法だと、各treeエージェントの判断は次のようになります。過去を参照する点以外、同じです。

```
Agt_Step{
Dim neighbor As Agtset
Dim one As Agt
```

```

If GetHistory(My.color, 1) == Color_Green Then
    MakeAllAgtsetAroundOwn(neighbor, T, False)
    For each one in neighbor
        If GetHistory(one.color, 1) == Color_Red Then
            My.color = Color_Red
            Break //もう調べる必要がない
        End if
    Next one
Else //1期前に燃えていても燃え尽きていても結局同じ
    My.Color = Color_Black
End if
}

```

このようにエージェントのルールを書き換えることで、同期させることができます。

それでは、上書き保存してから、実行してみましょう。実行した結果は、forest-fire(C)と同じように、期待した燃え方をしたでしょうか。

この方法は過去が現在に影響するので、「過去参照」法とか「因果的状態変化」法と呼べるかも知れません。

ところで、1期前の状態を参照して同期をとるとすると、第1ステップのときはどうなっているのでしょうか。それは、Univ_Init{ }でエージェントを生成するときに、colorの値を代入しています。これが第0ステップですから、第1ステップでGetHistory(My.color, -1)というルールを実行するときには、初期設定した色がMy.colorの1期前の値としてしまわれています。

新しく学んだ事項

- 同期問題
- 最後に一括置換する解決法
- バックアップ変数
- 過去を参照する解決法
- 変数に過去の値を記憶させておく方法
- 過去の変数の値を取り出す方法

- GetHistory()
- 繰り返し文を中止させるBreak

練習問題

本格的な山火事のモデルを作りましょう。

19.1

森林がある。最初、5%の確率で、燃えている木があり、その火が周囲の木に燃え移る。

まず準備作業から取りかかりましょう。Universeの下には空間(デフォルト)forestを作り、その下にtreeエージェント(エージェント数0)を作ります。treeには整数型変数colorを追加します。ここでは、GetHistory()を使うことにして、colorには記憶を1期分設定しましょう。また、森林の粗密(木の本数)をコントロールパネルで操作できるようにUniverseの下に実数型変数numberを追加します。マップ出力やコントロールパネルの設定をしましょう。なお、コントロールパネルからnumberを300から3000まで調節できるようにして下さい。



ルールの基本は、forest-fire(C)またはforest-fire(D)と同じで構いません。ただし、火が移る範囲は広め、たとえば1.5に設定しましょう。

19.2

延焼過程をグラフで示す。

森林火災のプロセスを把握するために、火勢と焼失率を調べるために、燃えている木と燃え尽きた木の本数を集計する変数を整数型で作ります(burningとburnt)。この変数そのものではなく、木の総数

に対する比率として、火勢と焼失率を時系列グラフの出力値にするように設定して下さい。



Univ_Step_End{}でやる作業と時系列グラフ設定でやる作業とに分けます。

19.3

燃えている木がなくなったらシミュレーションを終了させる。



全部燃え尽きたら、というわけではありません。

第20章

第2部の修了を記念して： コンウェイの「ライフゲーム」を作る

20.0 artisocのパワーを実感できます

- こんなに簡単にできてしまってよいのでしょうか
- 同期と簡単な場合分けを使うだけです
- 新しく学ぶ技法は全然ありません
- 少し工夫を加えましょう

20.1 ライフゲームとは

数ある人工生命的モデルの中でも、「ライフゲーム」は大変に有名なモデルです。數学者ジョン・コンウェイが考案したもので、セルオートマトン(セル)のルールを極限まで単純化し、各セルは0か1の2つの状態(生か死)しかとらず、状態が変化するかどうか(「状態遷移」と言います)は隣接する8セルの状態で決まるというモデルです。モデルができたのは、第1部の最後で作ったシェリングの分居モデルと同様に1960年代末のことです、そのときは人の手と石ころそしてテーブルや床を使ったのだそうです。このきわめて単純なルールからできているライフゲームは、まもなくコンピュータでも再現されます。

周囲のセルの状態と自分の状態とがどのような関係にあるのかは次節で説明しますが、要するに、セルは混雑していても過疎でも死んでしまい、適度な密度のときにしか生きられないという考え方です。その結果、一辺が2の正方形で安定したり、セルが3つ並んだ長方形が縦の状態と横の状態とを繰り返したり、さまざまな形状と変化を見せます。生きたセルがたくさんあると思ったつかの間全部死んでしまったり、少数の生きたセルの群がすぐに死ぬと思いきやしぶとく生き残ったりするので、予測も困難です。

と言ってもイメージが掴めませんね。「百聞は一見に如かず」とは、まさにこのことです。さっそく、コンウェイ

の発想をそのままモデル化しましょう。

artisocを使うと、モデルを驚くほど簡単に作ることができます。同期の問題に注意さえすれば、場合分けのルールしか用いません。あまりにも簡単すぎて、あっけないくらいです。

20.2 ライフゲームのモデル化

ライフゲームをartisocでモデル化する上での基本は、次のようにになります。

1. 平面にびっしりと格子型(セル型)のエージェント(以下、セル)を並べる
2. 各セルは生と死という2つの状態をとる
3. 各セルは、周囲(ムーア近傍)の8セルの状態に応じて、自分の状態を変える
4. 今、生きているセルは、周囲のセルのうち2または3個が生きていれば生き続けるが、それ以外の場合は死ぬ
5. 今、死んでいるセルは、周囲のセルのうち3個が生きているときのみ、生きる状態になる(誕生する)が、それ以外の場合は死んだままである

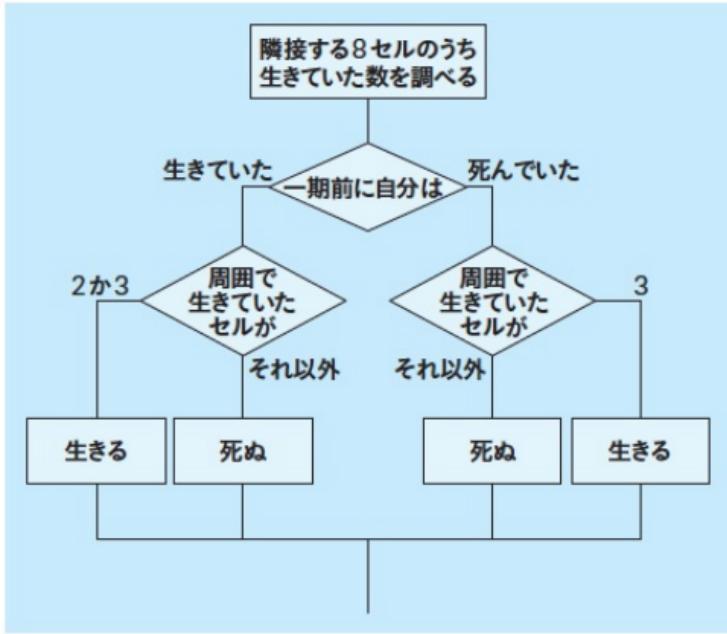


図20.1

では準備作業に取りかかりましょう。Universeの下に、 20×20 でループする空間fieldを作り、その下にエージェントcellunitを作ります。Universeのルールでエージェントを生成して空間に貼り付けるので、エージェント数を0にします。

ここでのモデル化では、過去参照法(GetHistory()の利用)により同期させることにします。セルの生死はマップ上の色で表すことにしましょう。そこで、cellunitに状態を表す整数型変数colorを追加します。colorには記憶を持たせるように設定して下さい。セルが生きていれば「水色」、死んでいれば「白色」にしましょう。

出力マップの設定に移ります。cellunitの出力では、エージェント表示色を変数指定とし、colorを選択

して下さい。マーカーに、■を選択すると、セルが敷き詰められている感じになります。また、マップのXとYの最大値を19にすると「のりしろ」が見えなくなります。準備作業が終わったら、game-of-lifeという名前で保存しましょう。

20.3 セルを貼り付ける

ライフゲームがどのような展開をたどるかは、初期状態によって決まります。ここでは、初期状態（各セルの生死）はランダムに決めることにしましょう。たとえば、下の作成例では、0.1の確率で「生」になるように初期設定しています。こうすると、生死の初期配置はいろいろなパターンになります。

```
Univ_Init{
Dim i As Integer
Dim j As Integer
Dim one As Agt
For i = 0 To 19
    For j = 0 To 19
        one = CreateAgt(Universe.field.cellunit)
        one.X = i
        one.Y = j
        If Rnd() < 0.1 Then
            one.color = Color_Cyan
        Else
            one.color= Color_White
        End if
    Next j
Next i
}
```

20.4 ライフゲームのロジック

各セルがどのような状態になるのかは、自分と周囲のセルの1期前の状態で決まります。以下のルールは、[図20.1](#)のフローチャートを忠実に記述しています。

```
Agt_Step{
Dim neighbor As Agtset
Dim one As Agt
Dim alive As Integer
alive = 0
MakeAllAgtsetAroundOwnCell(neighbor, 1, False)
```

```

For each one in neighbor //周囲の生きているセルを勘定
  If GetHistory(one.color, 1) == Color_Cyan Then
    alive = alive + 1
  End if
Next one
If GetHistory(My.color, 1) == Color_Cyan Then
  If alive == 2 Or alive == 3 Then
    My.color = Color_Cyan //生きたまま
  Else
    My.color = Color_White //死ぬ
  End if
Else //1期前、自分が死んでいれば
  If alive == 3 Then
    My.color = Color_Cyan //生き返る
  Else
    My.color = Color_White //死んだまま
  End if
End if
}

```

これで状態遷移のルール表記は完了です。

上書き保存して、いよいよ実行してみましょう。何回か繰り返して実行し、しばらく、ライフゲームの世界を楽しんで下さい(図20.2参照)。2, 3ステップで変化しなくなる場合もあれば、1000ステップ目でようやく変化しなくなる場合もあり、何回繰り返しても同じパターンが現れることはないでしょう。

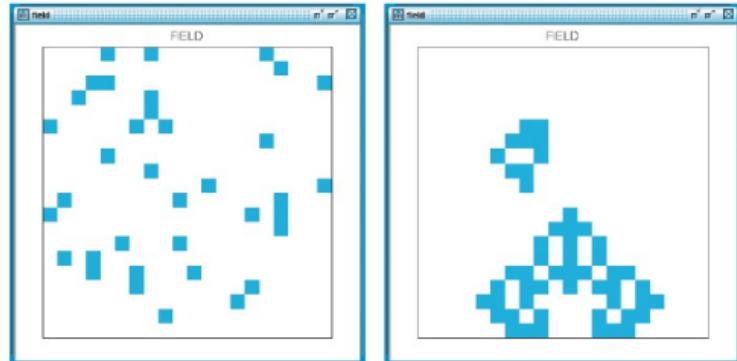


図20.2

初期状態の生死の分布が変化していく様子は、予測できそうでいてなかなか難しいです。単純ですが、状態遷移というライフゲームの本質が浮き上がります。運が良ければ、長い周期の興味深いパターンを見ることができますよ。

20.5 自動的に終了させる

何回か実行してみると、いろいろな経過をたどると思いますが、全てのセルが死んでしまう場合、生きているセルにも動きがなくなってしまう場合、あるいは単純な繰り返しが続く(周期2で反復する)場合に遭遇したことだと思います。そのような、見ても面白くない状態を見続けるのは退屈です。

そもそもいたん変化しない状態になってしまえば、その後、いくら待っても状態は変わりません。そのような状態になったらシミュレーションを自動的に終了させることにしましょう。

各ステップの最後に、全てのセルについて1期前の状態と現在の状態とが等しければ、もう何も変わりません。このような場合には、シミュレーションを終了させることにします。

```
Univ_Step_End{
Dim total As Agtset
Dim one As Agt
Dim stop As Boolean // ブール型の一時的変数
stop = True
MakeAgtset(total, Universe.field.cellunit)
For each one in total
  If one.color <> GetHistory(one.color, 1) Then
    stop = False
    Break // もう調べる必要がない
  End if
Next one
// Breakでここに出る
If stop == True Then
  ExitSimulationMsgLn("Stationary after" & GetCountStep() &
"steps")
End if
}
```

このルールで、全てのセルが死んだ場合、あるいは固定的な生きたセルの群しか残っていないようになった場合、シミュレーションは自動的に終了するようになります。正確にいうと、この記述は、どこか1つの

セルで1ステップ前の状態と現在の状態が異なれば、終了しない、というルールになっています。このようにルールを書くことによって、毎ステップ全てのセルの状態を調べることが不要になっています。

別の名前、たとえばgame-of-life(B)をつけて保存してから実行して、正しく自動終了するか確かめて下さい。

練習問題

20.1

同じ状態が周期2で反復するのをずっと眺めているのも退屈である。変化しない場合だけでなく、そのような場合も自動終了するようにルールを修正してみよう。一見複雑そうだが、実はとても簡単。



セルのうち1つでも、現在の状態が1期前とも2期前とも異なれば、終了条件が満たされません。

20.2

game-of-lifeモデルは、GetHistory()を用いた過去参照法で同期させている。このモデルを、一括置換法により同期させるようにルールを変えてみよう。



cellunitに新しい変数が必要になります。

20.3

Universeでのセルの初期配置を、入れ子構造のフォ文ではなく、RandomPutAgetsetCell()を使って、書き換えてみよう。



フォ文1つは必要です。

第3部

本格的な人工社会をめざす

人工社会を作る上でいろいろなアイデアが出てきても、シミュレータをうまく活用できないせいでモデルを構築できないようでは困ります。ここでは、artisocに備わっている機能をなるべく上手に活用して、さまざまなタイプの人工社会を作るための「道具」を紹介します。それらを活用して、アイデアを具体化して下さい。逆に、さまざまな道具の使い方を知ることによって、新しいアイデアが生まれてくることもあります。発想の源が増えることはよいことです。

この第3部のテーマは、関数の利用法、ルールの書き方、artisocのさまざまな設定の利用法など多岐にわたります。また、有名なモデルについても、第1部や第2部のように「修了を記念して」という特別な章を設けるのではなく、新しい技法を学ぶ過程で作っていきます。楽しみながら、複雑な技法をどんどん身につけていって下さい。

第21章

エージェントを空間上で複雑に動かす

21.0 動かし方にはいろいろあります

- 羊を追いかける狼をモデル化します
- 「ループしない」空間にも慣れましょう
- 狼にも羊にもさまざまな動きをさせてみます
- 組み込み関数を多用します
- 状況に応じてこの章のさまざまな技法を利用して下さい

21.1 動かし方のレパートリー

マルチエージェント・シミュレーションと一言で言っても、いくつかタイプがあります。エージェントがセルオートマトンとして格子状に並び、近傍のエージェントどうしが相互作用するタイプは、いろいろなモデルの基礎構造になっています。人工生命のみならず、グラフ(ネットワーク)にも使われています。このタイプでは、エージェントは動かず、1カ所に静止しています。これとは対極的なタイプが、空間上をエージェントが動き回るもので。従来は、動かないタイプのモデルが多くたのですが、人工社会ではエージェントの動くモデルが増えてきています。

この章では、空間の上でエージェントをさまざまに動かす技法を紹介します。全てのルール表記をマスターしてもらうというよりは、自分のモデル作りの際に適当に組み合わせができるように、動かし方のレパートリーを覚えてもらうのが、この章の狙いです。特に、組み込み関数を組み合わせることによって、エージェントに複雑で高度な動きをさせられることを実感して下さい。

以下では、草原の上で狼と羊を動かすさまざまなルールを書いてみると、エージェントの高度な移動法を学んでいきます。

21.2 モデルの大枠

いつものようにツリーに空間、エージェント、変数を加えていき、モデルの基本構造を作つておきましょう。Universeの下に空間plain(空間の大きさは100×100、「ループする」のまま)を作り、そこにエージェント種wolfとsheepを追加します。エージェントの数は両者ともとりあえず0のままにしておいてください。wolfには、1ステップあたりの移動速度を表す変数speed(実数型)を作つておきます。マップ出力設定でwolf、sheepが二次元マップに出力されるよう設定しておきましょう。

wolfsheepのファイル名で保存して下さい。以上で準備は完了です。草原の上で狼と羊をさまざまに動かしていきましょう。おなじみのForward()やForwawrdDirectionCell()が基本ですが、座標に関わる組み込み関数いろいろあります。それらを使いこなせば、かなり複雑な動きをエージェントにさせることができます。

21.3 速さや方向を変えながら動く

これまでに、Forward()やTurn()を用いてエージェントを動かすモデルを作りました。まずは復習です。最初に、エージェントがステップ毎に速度(速さや方向)を変化させながら移動するルールを作つてみましょう。

ここで、狼に登場願いましょう。プロパティ・ウインドウを開いてwolfの数を10とします。たとえば、wolfが1ステップ0.1の加速度でspeedを0.0から最速3.0まで徐々に加速しつつ、突進していくルールは、次のように書くことができます。エージェントの初期位置およびDirectionはAgt_Init{ }でランダムに決めておきます。

```
Agt_Init{
My.X = Rnd() * 100
My.Y = Rnd() * 100
My.Direction = Rnd() * 360
}
Agt_Step{
//最高速度まで加速
If My.speed <= 3 Then
    My.speed = My.speed + 0.1
End if
Forward(My.speed)
```

}

ステップ毎にエージェントが向きを変えながら動くルールも同じように作ることができます。たとえば、wolfエージェントが毎ステップ加速しつつ、左右30度の範囲でランダムに体の向きを変えながら移動するふるまいは、次の一文を上のルールのForward(My.speed)の直前に挿入すれば実現することができます。

```
//左右にランダムに揺れる  
Turn(Rnd() * 60 - 30)
```

このTurn()を書き込んでから上書き保存して、実行してみましょう。

21.4 ループなしの空間で移動する

ここまでルールを書いて実行ボタンを押すと、10頭のwolfが文字通り縦横無尽に空間を動き回る様がマップ上に現れるはずです。それでは、空間のプロパティで「端点の処理」を「ループしない」に変えた場合どうなるでしょうか。ループしない空間にしたモデルを、wolfsheep(B)という名前で保存して、実行してみましょう。空間に境界が設定されたことにより、上下左右の端点に至ったwolfはこれ以上前進することができなくなります。上のルールでは、ステップ毎にwolfの方向が変化するようになっているので、しばらくしたら再び前進を始める狼も出でますが、一般的には、このようにループしない空間では、エージェントに空間の境界を認識させ、端点に至った場合には別なルールを実行させる必要性が生じます。

自分が空間の端にいるかどうかは、Forward()を使って認識させることができます。実は、Forward()という関数の機能はエージェントを前進させるだけではありません。エージェントを前進させた上で、指定された距離を進めた場合には-1を、進めなかつた場合には進めなかつた分の距離を値とする特殊な関数なのです。たとえば、「Forward(5)」として前進させたところ、空間の境界にぶつかって実際には2.15しか進めなかつた場合、ルールの中では、Forward(5)という関数はさらに進むべき2.85という値になります。

このことを利用すると、たとえばループなしの空間においてwolfが草原の端に至った場合に反転(つまり180度回転)する動きは、次のようなルールで表現することができます。上のForward(My.speed)の部分を書き換えてみて下さい。

```
//前進するが、空間の端なら反対を向く
If Forward(My.speed) <> -1 Then //端にぶつかった場合
    Turn(180)
End if
```

上書き保存してから実行して下さい。

なお、うまく移動できたときには-1になり、そうでないときには進めなかった分の距離になるのは Forward()だけではなく、同じような前進系の関数(たとえばForwardX()など)は全て、このような仕様になっています。

21.5 目的地をめざす

移動には無目的なものから、特定の目的地や標的を目指すものまでさまざまなものが考えられます。ここでは、10頭のwolfが草原の中央(X=50, Y=50)を目的地として直進する動きをルール化してみましょう。

その際役に立つのが、関数GetDirection()です。これはXY平面上のAB二地点間の角度を計算する関数であり

```
GetDirection (AのX座標, AのY座標, BのX座標, BのY座標, 空間名)
```

というふうに書きます。この関数を利用すると、草原の中央を目指すwolfのルールは次のように書くことができます。GetDirection()をAgt_Init{ }ではなくAgt_Step{ }で用いているのは、目的地を通り越した場合にwolfの向きが反転するようにするためにです。wolfsheep(B)モデルをwolfsheep(C)という名前で保存し、Agt_Step{ }のルールを下記のものと差し替えて下さい。

```
Agt_Step{
//体を目的地の方向に向ける
My.Direction = GetDirection(My.X, My.Y, 50, 50, Universe.plain)
Forward(0.5)
}
```

上書き保存してから実行しましょう。

実際にシミュレーションを実行すればすぐにわかるように、上のルールで動くwolfは目的地を目指して接近はしますが、多くの場合、到達せずに、目的地の付近で行ったり来たりする結果になります。それは、目的地に近づいたら速度を落とし、目的地上で静止するという行動がwolfに備わっていないからです。そこでwolfの速さを目的地との距離に比例する形で変化させ、草原中央に達したら静止するようルールを書き換えてみましょう。

AB二点間の距離を求めるには、MeasureDistance()という関数を用います。この関数は

MeasureDistance (AのX座標, AのY座標, BのX座標, BのY座標, 空間名)

というふうに書きます。

この関数を使って距離に応じて速さを比例的に変化させるwolfのルールは次のようになります。なお、草原中央から最も離れている(距離約70)wolfの速さは3、中央に達したwolfの速さは0になるようになっています。

ここで、wolfsheep(D)という名前で保存して実行して下さい。

```
Agt_Step{
Dim_distance As Double
//目的地の方向を向き、目的地までの距離を測る
My.Direction = GetDirection(My.X, My.Y, 50, 50, Universe.plain)
distance = MeasureDistance(My.X, My.Y, 50, 50, Universe.plain)
//距離に応じて速度を決める
My.speed = 3 * (distance / 70)
Forward(My.speed)
}
```

なお、空間の中央にエージェントを移動させるだけならば、MoveToCenter()を用いることもできます。これは、エージェントを与えられた空間の中央に「瞬間移動」させてくれる関数です。エージェントを空間中央に密集させた状態でシミュレーションを始めるときは、Agt_Init{ }にMoveToCenter()と書くとよいでしょう(たとえば、第18章の深海魚のように)。

21.6 格子空間をランダムに移動する

ここでいったんwolfの数を0にし、かわってsheepの数を100にして、100匹の羊を草原に登場させましょう。wolfにとって草原はただ走り回るだけの連続的な空間でした。しかしsheepにとって草原は違った意味を持っていると考えましょう。すなわち、草原は、格子で仕切られた各セルが一区画の牧草地に対応する離散的な空間と考えるわけです。マップ出力の設定で罫線を表示させるのもいいでしょう。
wolfsheep(E)という名前で保存して下さい。

エージェントに格子状の空間を移動させる際には、Forward()のかわりにForwardDirectionCell()を用いることを第9章で学びました。ここではこの関数を使って、毎ステップ羊が牧草地をランダムな方向に一区画ずつ移動するルールを作成してみましょう。具体的には、sheepのルールは次のようなものになります。

```
Agt_Init{
My.X = Int(Rnd() * 100)
My.Y = Int(Rnd() * 100)
}
Agt_Step{
ForwardDirectionCell(Int(Rnd() * 8), 1)
}
```

ルールを書き込んだら、上書き保存してから実行して下さい。

21.7 追いかける

ここで、10頭のwolfをsheepの恐ろしい天敵として再び登場させることにします。wolfの基本的な動きは周囲10の範囲内にsheepを見つけると、それを標的にして体の向きを変え、加速をしながら直進していくことにします。標的に近接した場合、wolfは移動をやめ（速度を0にし）sheepを食べてしまいます。羊を食べる行為はDelAgt()でルール化しましょう。wolfの標的はステップ毎に決めるものとし、周囲に複数のsheepがいる場合はランダムに選んだ一匹を標的とします。ポイントになるのは、標的の方向に体を向けるという動きです。上で出てきたGetDirection()を使うこともできますが、これをより簡単に実現するためにTurnAgt()を使ってみましょう。この関数は、括弧内のエージェントのいる方向にエージェントを向かせます。

```

Agt_Step{
Dim num As Integer
Dim one As Agt
Dim set As Agtset
MakeOneAgtsetAroundOwn(set, 10, Universe.plain.sheep, False)
num = CountAgtset(set)
If num == 0 Then //標的がいなければぶらぶらする
    Turn(Rnd() * 60 - 30)
    My.speed = 0.5
Else //いれば
    one = GetAgt(set, Int(Rnd() * unm))
    TurnAgt(one) //体を標的方向に向ける
    //標的がすぐ近くなら、止まって捕食
    If MeasureDistance(My.X, My.Y, one.X, one.Y, Universe.plain) <
1 Then
        My.speed = 0
        DelAgt(one)
    //離れていれば、限界速度まで加速
    Elseif My.speed <= 3 Then
        My.speed = My.speed + 0.1
    End if
End if
//(捕食中以外)移動する(空間の端なら右を向く)
If Forward(My.speed) <> -1 Then
    Turn(-90)
End if
}

```

wolfsheep(F)という名前を付けて保存してから実行して下さい。

21.8 逃げる

天敵が登場したので、sheepの行動もそれに対応したものに変えましょう。具体的には、周囲にwolfを発見した場合、sheepは直近のwolfとは正反対の方向に一目散に逃げるように、ルールを変えてやります。周囲にwolfがない場合の行動は、21.6のルールと全く同じです。ポイントは、MeasureDistance()を使って周囲にいるwolfとの距離を逐一チェックして、最も近いwolfを選別している部分、直近のwolfとの角度をGetDirection()で取得して、180度を加えることで、反対方向に向きを変えている部分、および角度を格子空間における0から7の8つの方向に変換する部分です。

```

Agt_Step{
Dim degcell As Integer

```

```

Dim deg As Double
Dim distance As Double
Dim min As Double
Dim one As Agt
Dim threat As Agt
Dim set As Agtset
MakeOneAgtsetAroundOwn(set, 10, Universe.plain.wolf, False)
If CountAgtset(set) == 0 Then //天敵がないなら
    ForwardDirectionCell(Int(8 * Rnd()), 1) //ランダムな方向に移動
Else //いるなら直近の狼から一目散に逃げる
    //一番近い狼の特定
    min = 150 //最短距離を初期化(最短距離になり得ない数)
    For each one in set
        distance = MeasureDistance(My.X, My.Y, one.X, one.Y,
Universe.plain)
        If distance < min Then //oneとの距離が最短なら、置き換える
            threat = one
            min = distance
        End if
    Next one
    //一番近い狼と反対の方向に逃げる
    deg = GetDirection(My.X, My.Y, threat.X, threat.Y,
Universe.plain) + 180
    degcell = Int((deg + 22.5) / 45) //角度をセル型の方向に変換
    If degcell >= 8 Then
        degcell = 0
    End if
    ForwardDirectionCell(degcell, 3)
End if
}

```

このモデルは、wolfsheep(G)という名前で保存してから実行してみましょう。

21.9 追いかけ続ける

21.7のルールでは、wolfは1ステップ毎に標的を設定し直していました。以下では、wolfにさらに執拗さを加えて、一度狙いを定めたsheepをひたすら追い続けるように、ルールに変更を加えてみましょう。

今度は1ステップを超えて同一の標的を追い続けることになるので、ツリー上でwolfに、標的をリストアップするためのエージェント集合型変数targetを追加しておきます。この変数をうまく使ってwolfの行動ルールを次のように書きます。

```

Agt_Step{
Dim num As Integer
Dim one As Agt
Dim set As Agtset
If CountAgtset(My.target) == 0 Then //標的がいなければ周囲をさがす
    MakeOneAgtsetAroundOwn(set, 10, Universe.plain.sheep, False)
    num = CountAgtset(set)
    If num == 0 Then //いなければぶらぶらする
        Turn(Rnd() * 60 - 30)
        My.speed = 0.5
    Else //いれば狙いをつける
        one = GetAgt(set, Int(Rnd() * num))
        AddAgt(My.target, one) //oneを標的にリストアップ
    End if
Else //既に標的が決まっているれば捕食するまで追い続ける
    one = GetAgt(My.target, 0)
    TurnAgt(one) //体を標的の方向に向ける
    //標的がすぐ近くなら、止まって捕食
    If MeasureDistance(My.X, My.Y, one.X, one.Y, Universe.plain ) < 1 Then
        My.speed = 0
        TerminateAgt(one) //抹消する
    //離れていれば、限界速度まで加速
    Elseif My.speed <= 3 Then
        My.speed = My.speed + 0.1
    End if
End if
//(捕食中以外)移動する(空間の端なら右を向く)
If Forward(My.speed) <> -1 Then //前進し、端にぶつかったら右を向く
    Turn(-90)
End if
}

```

wolfsheep(H)という名前で保存して下さい。

上のルールでAddAgt()とTerminateAgt()は初出です。

AddAgt()は特定のエージェントをエージェント集合型変数に追加してリストアップする関数です。これにより、標的が決まります。

TerminateAgt()は括弧内エージェントを瞬時に抹消しますが、第16章やこの章のwolfsheep(F)モデルで用いたDelAgt()よりも徹底していて、それ以外のエージェントの情報(記憶)に抹消されたエージェン

トがリストアップされている場合には、それからも抹消します。つまり、空間上からただちに消える(DelAgt()の機能と同じ)だけでなく、エージェント集合型変数からも除去するのです。このようにTerminateAgt()は他のエージェントについてもエージェント集合型変数を逐一調べるという大量の仕事をするために若干動きが遅くなるので、そうする必要のあるときだけ使うように心がけて下さい。詳しくは、コラム「[エージェントを消す三つの関数](#)」を参照して下さい。

上のルールで、DelAgt()を使ってエージェントを消すと、もし他の狼も同じ羊を追っていた場合には、その狼にとっては羊がまだ存在している(My.targetの中に残存している)ままです。それでは具合が悪いので、TerminateAgt()を使う必要があるのです。TerminateAgt()を使うこととエージェント集合型変数targetの使い方とは密接に関連しているのです。逆に言うと、wolfsheep(F)では、このような問題が生じないので、DelAgt()でかまわなかったのです。

21.10 座標系に関連づけて移動する(とばしてもかまいません。)

以上、単純なものから複雑なものまでさまざまな移動のパターンを紹介してきましたが、エージェントの移動そのものはForward()ないしForwardDirectionCell()という組み込み関数を使ってルール化してきました。これに対して、エージェントのX座標、Y座標を直接操作して移動させることもできます。

そもそも、エージェントが空間を動くということはエージェントの位置が変化するということです。ある場所から与えられた方向に等速直線運動する場合を考えましょう。artisocのルールで書こうするとと、My.speedとMy.Directionの値が既に与えられているものと仮定すると

```
My.X = My.X + My.speed * Cos(DegreeToRad(My.Direction))  
My.Y = My.Y + My.speed * Sin(DegreeToRad(My.Direction))
```

が座標の変化の基本ルールになります。これに加えて、ループしている場合といない場合とで異なる端での座標の値の処理ルールが必要になります。コサイン(Cos())とかサイン(Sin())とか三角関数が出てくるだけで、もういやになってしまったかもしれません。三角関数の中に出てくるDegreeToRad()は角度の単位をラジアンに変換するための関数です。等速直線運動という最も単純な場合でさえこれから、複

雑な動きをルール化するのがいかに面倒かわかるでしょう。

しかし、Forward系関数がエージェントの現在位置を基準に移動を実行するのに対して、座標値を直接操作する移動方法は、エージェントをチェス駒のように動かす「神様の視点」からの移動の実行方法と言うことができます。このような移動方法は、ある決まった軌道に沿ってエージェントを移動させたいときなどに役に立ちます。

たとえば、wolfに、草原の中央を中心にして、その周りをさまざまな速さでぐるぐる周回させて、円を描かせたいときには、次のようにルールを書くといいでしよう。wolfに実数型変数degreeを追加して、wolfsheep(I)という名前で保存して下さい。degreeは草原の中央を原点に取ったときのエージェントの座標とX軸がなす角度を表します。なお、このモデルでは、speedはdegreeが変化する幅、つまり1ステップあたりの角速度になります。

```
Agt_Init{  
    My.degree = 360 * Rnd()  
    My.speed = 10 * Rnd()  
}  
Agt_Step{  
    My.degree = My.degree + My.speed  
    My.X = 50 + 30 * Cos(DegreeToRad(My.degree))  
    My.Y = 50 + 30 * Sin(DegreeToRad(My.degree))  
}
```

以上です。上書き保存してから、実行してみましょう。狼は円運動をしているでしょうか。

新しく学んだ事項

- ループしない空間の端点での移動の扱い
- Forward()の値の利用
- GetDirection()
- TurnAgt()
- MeasrueDistance()
- AddAgt()

- `TerminatgeAgt()`
- 最小値の探し方(初期化, フォ・イーチ文, イフ文, 置き換え)
- 三角関数の初步的な使い方

練習問題

21.1

ループしない空間で狼を動かす`wolfsheep(B)`モデルでは、端にぶつかった狼は動く向きを反転させただけだった。そこで、もう少し複雑なルールにしよう。

端にぶつかったら、反転して、残りの距離を進む。



Forward(`My.speed`)を一時的変数に代入しておくと便利です。

21.2

21.8で`woldsheep(G)`を作る際、最も近い狼から逃げるルールを学んだ。ポイントになった技法は最小値の求め方である。この技法の応用を試みる。`wolfsheep(F)`では、狼は、追いかける羊をランダムに選んでいる。そこで、もっと現実的なルールにしよう。

狼は、一番近くにいる狼を追いかけて、捕食する。



一時的変数が全然足りません。

エージェント集合の中のエージェントの並び方

MakeAgtset()やMakeAllAgtsetAroundOwn()などの関数を使ってエージェント集合を作ったときに、そこに含まれる各エージェントに0から $n-1$ までの番号(n は集合の要素数)が割り振られていることは、第14章でGetAgt()関数を説明したときに触れました。では、artisocはどのような規則にしたがってエージェントに番号を付けているのでしょうか？ 実は、一般的には、そのような規則は存在しません。つまり、エージェント集合に含まれる特定のエージェントが、集合の中で何番目に位置づけられるのかをあらかじめ知ることはできないのです。わかっているのは、その番号が0から $n-1$ の範囲のある値であるということだけです。

それでは、この番号は、「GetAgt(set, Int(Rnd() * n))」というおなじみの乱数指定による使い道しかないのでしょうか。実は、エージェント集合に操作を加えることで、この番号を制御可能にする方法が二種類存在します。ひとつは、第30章で出てくるSortAgtset()でエージェントをソートする方法です。ある変数値にしたがって降順・昇順にエージェント集合の中のエージェントを並び替えると、エージェントの番号がその順番にしたがった形に振りなおされます。たとえば、「SortAgtset(set, "score", False)」とした上で「GetAgt(set, 3)」とすると、変数scoreの値が四番目に大きいエージェントを取り出すことができます。

もう一つの方法は、AddAgt()を使って、初めから(つまりMakeAgtset()などの関数に頼らずに)自分でエージェント集合を作る場合です。フォーチュンでエージェントをめぐって適宜選別し、「AddAgt(set, one)」などとして別のエージェント集合に加えていくと、加えられた順番にエージェントの番号が0から割り振られていきます。エージェントを追加する順番をさまざまに工夫することで、任意の番号を任意のエージェントに割り当て、活用することができます。

しばしば犯すミスは、たとえばGetAgt(S, n)などとすることにより、GetAgt()関数の中で、対応するエージェントが存在しない要素番号nを指定してしまうことです。特に、エージェント集合型変数Sが、エージェントを全く含まない空集合の場合、要素番号自体が存在しないため、nにどのような値を指定してもエラーが発生してしまいます。第14章で、GetAgt()関数を使用するにあたって、エージェント集合の要素数を数えた上で空集合のケースを排除したのも、このようなエラーの回避を念頭に置いてのことでした。

第22章

属性を文字列で表す

22.0 質的な属性を質的なままで操作できます

- 属性を表す変数は文字列が基本です
- 一つの変数でいくつもの属性を表せます
- 属性は、文字としての数字で表すと便利です
- 数字で表された属性は、色の属性に簡単に変換できます
- 属性の中身(一部分)を取り出したり、変えたりすることもできます

22.1 文字列で表現するとは

人間の属性には年齢、体重、身長といった数字で表せるものと、性別、国籍、信教、母語といった質的で数字では表せないものとがあります。本書のモデル作りでは、質的な属性は、たとえば第13章では、色(健康人ならColor_Cyan、病人ならColor_Red)で区別してきました。

この章では、属性を文字列で表すことにより、複雑な属性を扱うさまざまな技法を学びます。たとえば、性別という属性を男とかmale、女とかfemaleと表す方法です。遺伝子DNAならAGGCTTAACなどと表すこともできます。その代わり、数の演算(+−×÷)とは異なる演算が必要になるので、文字列を扱う新しい操作方法が登場します。

さらに、質的な属性を色で表す技法と文字列で表す技法とを結びつける手法も学びましょう。数字が整数としての意味と文字としての意味とを持ちうるという二重性をうまく利用します。

22.2 文字列型の変数に慣れる

artisocでは文字列の操作は、変数を文字列型に設定して行ないます。文字列型の変数の中には数字を入れることもできますが、あくまで文字としての数字です。普通の変数(たとえば整数型)と文字列型変数とでは、次のように違います。

- 整数型変数x:「 $x = 1 + 2$ 」は $1 + 2$ という足し算の答えをxに代入することですから、xの中を覗いてみると3が入っています。
- 文字列型変数y:「 $y = 1 + 2$ 」という代入文はエラーになります。その代わり、「 $y = "1 + 2"$ 」と表記すると、yには文字列 $1 + 2$ が入っています。「 $y = "1" & "2"$ 」と書くと、文字列12が入ります。文字列を操作するには、引用符「“.....”」と記号「&」が重要です。これらの記号自体は、第1部(第7章)で初めて使って以来、何度も使っているので、おなじみですね。

これから、文字列の操作を学びます。最初は、とても簡単なモデルを作りましょう。

広い部屋に人がたくさん歩き回っている。近くに同性の人がいると、その人と同じ方向に歩き出す。

Universeの下に空間(デフォルト)roomを作り、その下にエージェント種person(エージェント数200)を作ります。personには性別とそれをマップに色別に図示するための変数、trait(文字列型)とview(整数型)を作ります。Agt_Init{ }では、次のように初期設定します。ランダム変数を利用して、男女がだいたい半々になるようにします。男のtraitはmaleという値、viewは赤色という値をとり、女はfemaleと青色です。マップ出力も適切に設定しておきます。ここまでのお作業をpartyという名前で保存しましょう。

```
Agt_Init{
My.X = Rnd() * 50
My.Y = Rnd() * 50
My.Direction = Rnd() * 360
If Rnd() >= 0.5 Then
    My.trait = "male"
    My.view = Color_Red
Else
    My.trait = "female"
    My.view = Color_Blue
End if
}
```

personエージェントは次のようなルールで行動します。ひとつはぶらつくこと、もう一つは、もし近くに同性の人がいたらその人と同じ方向に歩き出すこと、です。

```
Agt_Step{
Dim neighbor As Agtset
Dim one As Agt
Dim num As Integer
Turn(Rnd() * 60 - 30)
Forward(1)
MakeAllAgtsetAroundOwn(neighbor, 1, False)
num = CountAgtset(neighbor)
If num > 0 Then
    one = GetAgt(neighbor, Int(Rnd() * num))
    If My.trait == one.trait Then
        My.Direction = one.Direction
    End if
End if
}
```

文字列型変数でも、その値(つまり文字列自体)が同じかどうかを比べて判断するときには、If My.trait == one.trait Thenのように、「==」を使います。

上書き保存してから実行して下さい。

実は、このモデル作りは既に第12章で学んだ技法ではもっと単純なルールでした。色を表す変数だけを用いて

```
If My.view == one.view Then
    My.Direction = one.Direction
End if
```

というルールを書けば同じになりますね。しかし、ここでは、文字列型変数を利用する練習だと思って、我慢して下さい。以下で、このルール表記を基にして、複雑なモデルを作ってもらいます。

22.3 文字列型変数も演算(操作)の対象になります

上の例では1つの属性(性別)を1つの文字列型変数で表しましたが、複数の属性を1つの変数で表す

ことも可能です。たとえば、最初の文字で性別(男ならm女ならf)、次の3文字で国籍(日本ならJPNアメリカならUSA)、最後の文字で働いているかどうか(有職ならe無職ならu)とすると、fJPNeは「職に就いている日本女性」という属性になります。

属性を表す変数traitに、「職に就いている日本女性」という属性を表す値を与えるときには、たとえば

```
My.trait = "f" & "JPN" & "e"
```

というふうに代入します。もし、すでにMy.traitにfJPNという値がしまわっているとき、

```
My.trait = My.trait & "e"
```

という演算をすると、My.traitはfJPNeになります。

さて、このような属性を表す変数traitがあるとき、変数の一部分を取り出す必要が生じることがあります。たとえば、エージェント同士の国籍を比較する場合、ある変数の値と別の変数の値とが同じかどうか(つまり同じ文字列かどうか)比べる技法だけでは不十分で、真ん中の三文字だけを取り出して、部分と部分とを比べる必要が出てくるわけです。

artisocでは、文字列型変数について、全体の文字列から一部分だけを取り出す方法がそろっています。

- 左右の端から取り出す: Left()もしくはRight()を使います。()の中には、対象文字列、長さ、の二つを指定します。たとえば、Left(My.trait, 1)と書けばMy.traitの左端から1文字、Right(My.trait, 2)と書けば右端から2文字を表します。
- 真ん中から取り出す: 両端でなく、任意の場所から文字列の一部分を取り出す場合には、Mid()を使います。今度は開始位置も指定してあげる必要がありますから、()の中には、対象文字列、開始位置、長さ、の3つを指定します。もしMy.traitの国籍を取り出すのであれば、二文字目から三文字取り出すのですから、Mid(My.trait, 2, 3)と書きます。

とくに真ん中から取り出す方法は、左からの場合も右からの場合も使えるので、少し複雑ですが万能です。たとえば、`Mid(My.trait, 1, 1)`とすれば左端から一文字取る事ができるので、`Left(My.trait, 1)`と同じ操作になります。`artisoc`では、0から数えるのが基本ですが、文字列の最初(左端)は0ではなく1なので、注意して下さい。

なお、文字列型変数の値、つまり文字列自体は半角文字が基本です。全角文字でもかまいませんが、その場合は1文字を2文字として扱う必要があります。文字列を操作する関数を全角文字列に対して用いるときには注意して下さい。

22.4 複雑な属性を異なった色で表す

属性を表す文字列変数は、1つの変数で、1つの属性(たとえば性別)だけではなく、多くの属性を表すことができます。上の例のように、男女、日米、有職・無職の組み合わせだと、3つの属性の8種類の異なるパターンを表せます。他方で、属性を色で表す方法もあります。この場合、8種類のパターンのひとつひとつに、異なる色を指定すれば、マップ上で、個々のエージェントの属性を識別することが可能になります。しかし、長いイフ文を書いて、いちいち対応させなければなりません。また、属性のとりうる値を大きくすると(たとえば国籍に中国とフランスを加える)、色を指定する作業が大変になります。

そこで、一方では、属性を数字の文字列で表し、他方で、属性を表す数字の文字列を、色を表す整数として利用するのです。上で指摘したように、数字はそのままでは文字列型変数には使えませんし、文字列の数字はあくまで「数」ではなく「字」です。そこで、数と文字との間を行ったり来たりする必要が出てきます。`artisoc`では次のように数と文字との変換(Convert)をすることが可能です。

- 整数から文字列へ:`CStr()`を使います。括弧の中には、整数(半角の数字)を入れます。たとえば、`CStr(7)`は、7という整数ではなく、文字としての7です。
- 文字列から整数へ:`CInt()`を使います。`CInt(7)`により、7という文字は、数としての7になります。

なお、`CStr()`, `CInt()`は整数と文字列との間の変換だけでなく、さまざまな型の変数を、それぞれ文字列や整数に変換するのに使うことができます。では、次のようなモデルを作りましょう。

性別(男女), 食べ物の嗜好(和洋中), 飲み物の嗜好(酒, ワイン, ノンアルコール)の3属性を持っているエージェントがパーティー会場でたくさん歩き回っている。近くに異性のエージェントがいて、飲食の嗜好が同じだと、同じ方向に歩き出す。飲食のどちらか一方だけの嗜好が同じだと相手の嗜好に合わせる。

ここで、性別を0, 1, 飲食の嗜好を各々0, 1, 2で区別することにしましょう。そして、属性のパターンにしたがってエージェントを異なる色で表示しましょう。上で作ったpartyモデルを修正します。そこで、party(B)という名前を付けて保存してから作業しましょう。

まず、Agt_Init{ }のルールでは、My.traitとMy.viewの設定を次のように変えます。これは、初期値をランダムに与える設定方法です。

```
Agt_Init{
My.X = Rnd() * 50
My.Y = Rnd() * 50
My.Direction = Rnd() * 360
My.trait = CStr(Int(Rnd() * 2)) & CStr(Int(Rnd() * 3)) &
CStr(Int(Rnd() * 3))
My.view = RGB(CInt(Mid(My.trait, 1, 1)) * 240, CInt
(Mid(My.trait, 2, 1)) * 120, CInt(Mid(My.trait, 3, 1)) * 120)
}
```

ここで、CStrとCIntの使い方に注意して下さい。たとえば、CStr(Int(Rnd() * 2))は、文字としての0か1になります。CInt(Mid(My.trait, 1, 1))は、traitの左端の1文字(0か1)を整数としての0か1に変換します。CInt(Mid(My.trait, 3, 1))は右端について同様の作業をします。

RGB関数の使い方は既に第12章で学びましたね。RGB(255, 255, 255)にならないように、少し工夫がしてあります。なぜでしょう？ 理由が分かりますか？

次に、Agt_Step{ }のルールを書き込みましょう。大枠は上の単純なモデルと同じです。ただ、上のモデルだと同性の場合に同じ方向に歩き出すルールが、ここでは異性だといろいろな判断をエージェントがするルールになります。一応フローチャートでエージェントの行動ルールを確認しましょう。

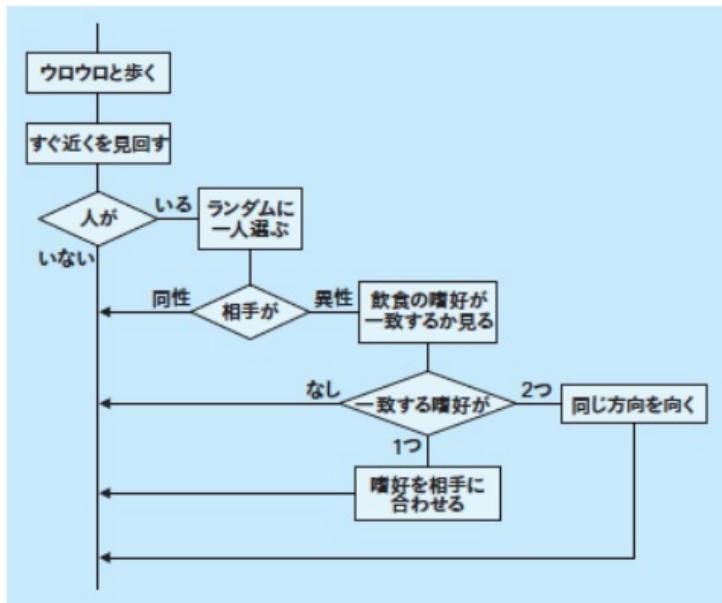


図22.1

```

Agt_Step{
Dim neighbor As Agtset
Dim one As Agt
Dim num As Integer
Dim i As Integer
Dim s As Integer
s = 0 //嗜好の類似度sを初期化
Turn(Rnd() * 60 - 30)
Forward(0.5)
MakeAllAgtsetAroundOwn(neighbor, 1.5, False)
num = CountAgtset(neighbor)
If num > 0 Then
    one = GetAgt(neighbor, Int(Rnd() * num))
    If Left(one.trait, 1) <> Left(My.trait, 1) Then //異性ならば
        For i = 2 To 3
            If Mid(one.trait, i, 1) == Mid(My.trait, i, 1) Then
                s = s + 1
            End if
    End if
}

```

```

Next i
If s == 2 Then    //嗜好が一致のとき
    My.Direction = one.Direction
Elseif s == 1 Then    //飲食どちらかの好みが一致
    My.trait = Left(My.trait, 1) & Right(one.trait, 2)    //相手
の嗜好に合わせる
    My.view = RGB(CInt(Mid(My.trait, 1, 1)) * 240,
CInt(Mid(My.trait, 2, 1)) * 120, CInt(Mid(My.trait, 3, 1)) *
120)
        End if
    End if
End if
}

```

上のルールでは、Left(), Mid(), Right()が登場しました。何をさせているのか、わかりますか。フローチャートとルールとを照らし合わせて考えて下さい。

上書き保存の上、実行しましょう。エージェントの色や動く方向の変化に注目して下さい。

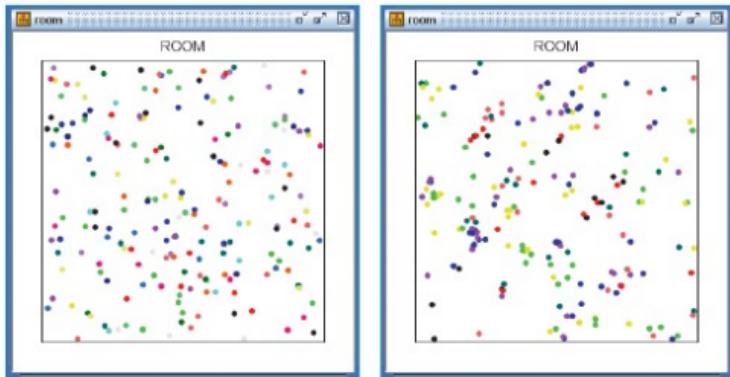


図22.2

新しく学んだ事項

- 文字列型変数

- 文字列からその一部分を抽出する関数Left(), Mid(), Right()
- 整数型変数と文字列型変数との間で型を変える関数CInt(), CStr()
- 属性と色との技巧的な関連づけ

練習問題

エージェントの属性(の違い)を色(の違い)で表すことだけでなく、一般的にさまざまな変数値を色で視覚化することは、シミュレーションを直感的に理解する上で重要な技法です。変数値と色との関連づけに慣れておきましょう。

22.1

パーティーのモデルparty(B)では、エージェントの属性は次のように与えられている。

```
My.trait = CStr(Int(Rnd() * 2)) & CStr(Int(Rnd() * 3)) &
CStr(Int(Rnd() * 3))
My.view = RGB(CInt(Mid(My.trait, 1, 1)) * 240,
CInt(Mid(My.trait, 2, 1)) * 120, CInt(Mid(My.trait, 3, 1)) *
120)
```

属性は全部で18パターンある。それぞれ、何色に表示されるだろうか？

22.2

party(B)モデルでは、パートナーになった相手は異性なので、嗜好が同じでも、マップ上では同色にならない。そこで修正を加えてみよう。

同じ方向を向くことにしたエージェントは性別の違いにかかわらず、共通の嗜好に応じて同じ色で表示する。



traitは変えません。viewに代入するRGB関数の表記を工夫します。

第23章

属性を複雑に操作する：アクセルロッドの文化変容モデル（簡略版）を作る

23.0 周囲の状況に応じて、文化が複雑に変化します

- 前章で学んだ文字列操作法の応用が中心です
- 文字列の操作に慣れましょう
- 新しい技法も学びます
- 多様性を調べる技巧的な方法を教えます
- アクセルロッドの「文化変容」モデルの簡略版を作ります

23.1 文字列の書き換えを文化変容に対応させる

文字列型変数を用いたモデル作りに慣れましょう。人工社会で、属性を用いるモデルの典型は文化をめぐる相互作用です。特定の文化（属性の組み合わせ）をもった集団（共同体）が周囲の集団と相互作用する中で、文化の伝播、流布、均質化などが進みます。

文化をめぐる相互作用をモデル化したものとして有名なのが、ロバート・アクセルロッドによる「文化変容」モデルです。以下では、アクセルロッドのモデルの本質を再現しつつも、少し単純化したモデルを作りましょう。

ある社会には、文化的に多様な共同体が共存している。周囲の共同体との文化の類似性に応じて、自分の文化が周囲の共同体の文化に似てくる。

ではモデル化の準備作業に取りかかりましょう。

1. Universeの下に空間country(大きさを 10×10 にし、それ以外はデフォルト)を追加し、それにエージェントcommunityをエージェント数0(デフォルト値)で追加します。
2. communityエージェントには、文化を表す文字列型変数traitと文化に応じた色を表す整数型変数viewを追加します。
3. マップ出力の設定をします。要素設定ではエージェントの色指定を忘れずにして下さい。また、最大値を10ではなく9にしておくと、「のりしろ」が隠れて見栄えがよくなります。
4. このモデルをdisseminationという名前で保存して下さい。

23.2 エージェントの初期配置

このモデルでは、集団(共同体)は動き回らず、村落のように国土に張り付いているものと考えるので、格子型としてエージェントを初期配置することにします。

なお、エージェントは格子型の空間に敷き詰めるので、 10×10 の大きさの空間では100生成すればよいわけですが、空間の大きさを柔軟に変えられるように、以下のルールでは新しい関数GetWidthSpace()、GetHeightSpace()を用いています。これらは、想像できると思いますが、空間の大きさ(幅と高さ)をルールの中で測定する関数です。

```
Univ_Init{
Dim i As Integer
Dim nation As Agtset
For i = 0 To GetWidthSpace(Universe.country) *
GetHeightSpace(Universe.country) - 1
    CreateAgt(Universe.country.community)
Next i
MakeAgtset(nation, Universe.country.community)
RandomPutAgtsetCell(nation, False)
}
```

For $i = 0$ To 99とする代わりに、上のように書き込むと、空間の大きさをプロパティ・ウインドウで変えても、ルールを書き換える必要がありません。なお、マップ出力の範囲も、ダイアログを通して、簡単に空間の大きさの変更に対応させることができます(のりしろをなくすのは手作業です)。

各エージェントに属性(と色)を割り当てるのは、エージェントのルールエディタで行ないます。ここで、文化は3つの側面(文化要素)から構成され、各々0から9までの10種類の異なるタイプからなる—たとえば、母語(10言語)、宗教(10宗派)、職業(10種類)—と仮定しましょう。属性はランダムに割り振ることにします。

```
Agt_Init{  
My.trait = CStr(Int(Rnd() * 10)) & CStr(Int(Rnd() * 10)) &  
CStr(Int(Rnd() * 10))  
My.view = RGB(CInt(Mid(My.trait, 1, 1)) * 28, CInt(Mid(My.trait,  
2, 1)) * 28, CInt(Mid(My.trait, 3, 1))*28)  
}
```

属性を色に変換する際、CInt()*28となっていますが、なぜ28なのかを考えて下さい。

とりあえず、上書き保存をして、実行ボタンを押して下さい。さまざまな色(属性)のエージェントがきっちりと並んでいるでしょうか([図23.1](#))。

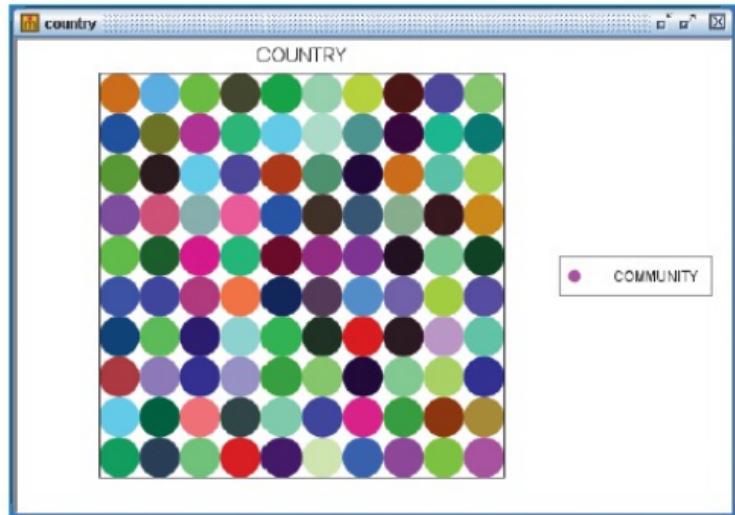


図23.1

23.3 周囲のエージェントの属性に応じて自分の属性を変える

エージェントの属性変化のルールのポイントは、(1)近くにいるエージェントとの文化の類似度を判断すること、(2)類似度の違いに応じて文化変容のパターンが異なることです。ここでは、文化が似ているほど、真似る確率も高くなります。

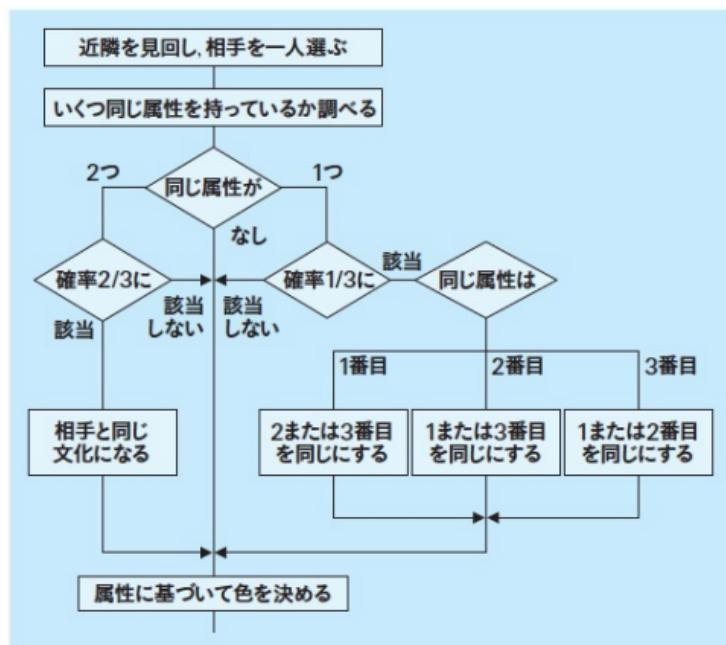


図23.2

```
Agt_Step{  
Dim neighbor As Agtset  
Dim one As Agt  
Dim i As Integer  
Dim s As Integer  
s = 0  
MakeAllAgtsetAroundOwnCell(neighbor, 1, False)  
one = GetAgt(neighbor, Int(Rnd() * CountAgtset(neighbor))) //近
```

```

隣から1つ選ぶ
For i = 1 To 3
  If Mid(one.trait, i, 1) == Mid(My.trait, i, 1) Then
    s = s + 1
  End if
Next i
If s == 2 and Rnd() * 3 >= 1 Then //類似度2なら確率2/3で変化
  My.trait = one.trait
  My.view = one.view
Elseif s == 1 And Rnd() * 3 >= 2 Then //類似度1なら確率1/3で変化
  If Mid(My.trait, 1, 1) == Mid(one.trait, 1, 1) Then //最初の属性が同じ
    If Rnd() < 0.5 Then
      My.trait = Mid(My.trait, 1, 1) & Mid(one.trait, 2, 1) &
Mid(My.trait, 3, 1)
    Else
      My.trait = Mid(My.trait, 1, 1) & Mid(My.trait, 2, 1) &
Mid(one.trait, 3, 1)
    End if
  Elseif Mid(My.trait, 2, 1) == Mid(one.trait, 2, 1) Then //2番目の属性が同じ
    If Rnd() < 0.5 Then
      My.trait = Mid(one.trait, 1, 1) & Mid(My.trait, 2, 1) &
Mid(My.trait, 3, 1)
    Else
      My.trait = Mid(My.trait, 1, 1) & Mid(My.trait, 2, 1) &
Mid(one.trait, 3, 1)
    End if
  Else //最後の属性が同じ
    If Rnd() < 0.5 Then
      My.trait = Mid(one.trait, 1, 1) & Mid(My.trait, 2, 1) &
Mid(My.trait, 3, 1)
    Else
      My.trait = Mid(My.trait, 1, 1) & Mid(one.trait, 2, 1) &
Mid(My.trait, 3, 1)
    End if
  End if
End if
My.view = RGB(CInt(Mid(My.trait, 1, 1)) * 28, CInt(Mid(My.trait,
2, 1)) * 28, CInt(Mid(My.trait, 3, 1)) * 28)
}

```

類似度が1のとき、表記が複雑に見えますが、Mid()を機械的に用いて、文字通り、機械的にルールを書いてあります。

それではdissemination(B)という名前で保存をして、実行しましょう。徐々に、文化が平準化していくこ

とが観察できるでしょうか(図23.3参照)。

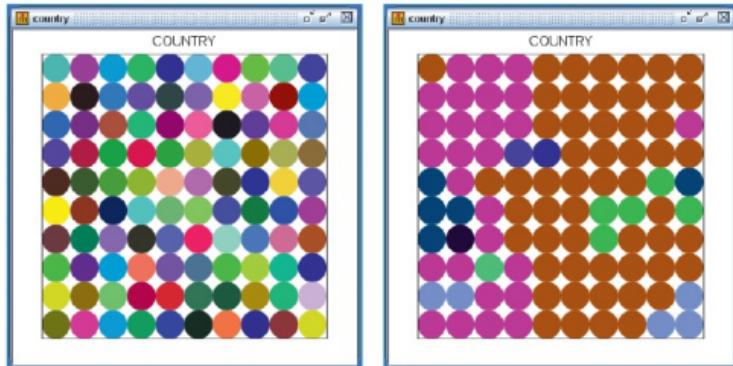


図23.3

23.4 オリジナルとの違い

ここで作った文化変容モデルは、アクセルロッドの文化変容モデルの発想法を忠実に再現しています。ただし、彼が作ったモデルに比べて3点で簡略化してあります。

まず、アクセルロッドのモデルでは文化を表す属性が、3つではなく、5つです。つまり、traitは5桁の数字からなる文字列です。しかし私たちのモデルでは、RGB関数を利用して簡単に色を表す方法をとっているために、3桁にしてあります。ルールの基本には何の違いもありませんが、文化を5桁にすると、類似度が6段階になるのでルールが複雑になるだけでなく、一致していない属性を選び出して変化させるルールも複雑になります。

第2の違いは、近隣エージェントの捉え方です。アクセルロッドのモデルでは、近隣にいるエージェントは、東西南北の4方向にいるエージェントだけです。これに対し、私たちのモデルでは、斜め方向も含めた8方向にいるエージェントを近隣のエージェントと考えました。つまり、アクセルロッドは「ノイマン近傍」で、私たちは「ムーア近傍」で近隣を定義しているのです(閑話休題[見方を変えると周りも変わる](#)を参照)。

artisocの組み込み関数は、ムーア近傍の考え方を採用しているので、このようにしました。

最後に、アクセロッドのモデルでは、空間はループしていません。つまり、空間の端では、近隣エージェントが少なくなっています。私たちのモデルは、空間がループしているので、エージェントの近隣関係は均質的で、どこにいるエージェントでも8エージェントと接しています。第2の相違点も最後の相違点も、比較するためにランダムに選ばれるエージェントの範囲が異なるだけで、本質的な違いではありません。

23.5 属性を直接マップに示す

今まで、エージェントの属性を色としてマップに表示する技法を用いてきました。たしかに複雑な属性を色で表示すると見栄えがきれいです。文化変容のモデルでは、エージェントの色が徐々に変化していくことで、文化が変わっていくことが実感できます。しかし、属性（文化）そのものがどうなっているのか知るには、色の表示はあくまで間接的です。

artisocは、エージェントの変数の値を直接マップの上に表示させることができます。この機能を使って、各エージェントがどのような文化パターンになっているのかを、色の表示だけでなく、直接その三桁の文字列を表示させることにしましょう。

マップ出力設定の編集作業をします。要素設定ウィンドウを開いて下さい。そこにエージェント情報という項目があります。まず、「情報表示」をチェックして、エージェントの変数リストからtraitを選択します（[図23.4参照](#)）。これで完了です。



図23.4

マップ出力設定を変えたモデルは、dissemination(C)という名前を付けて保存しましょう。実行してみると、各エージェントの上に文字列が表示されるはずです(図23.5参照)。

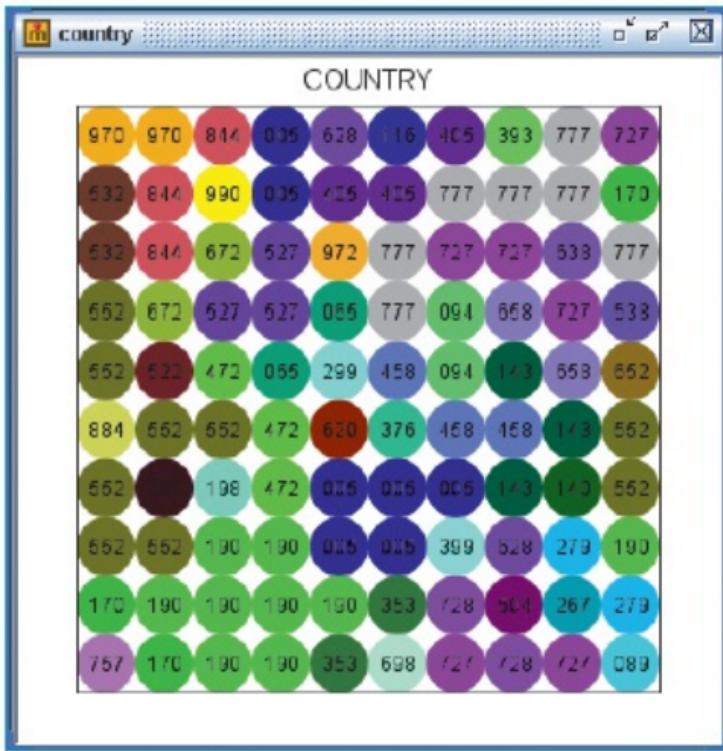


図23.5

23.6 全体の様子を把握する

最後に、個々の集団(共同体)の文化変容により、社会全体の文化的多様性がどのように変化するかを調べることにしましょう。

エージェントがさまざまなタイプの文化(属性の組み合わせ)を持っているときに、全部で何パターン存在しているのか調べる。

この技法は、属性の多様性を調べるときの基本です。是非、習得しましょう。一般的にいえば、ある文字列と同じものが別の文字列の中に存在しているかどうかを調べる技法です。

まず、Universeに文化の数を表す整数型変数numculを追加します。Univ_Step_End{ }で必要な集計作業をさせます。考え方の基本は、各エージェントの文化を次々ととってきて、既存の文化のレポートリーの中に同じものがあるか調べ、なければレポートリーに追加していく、ということです。

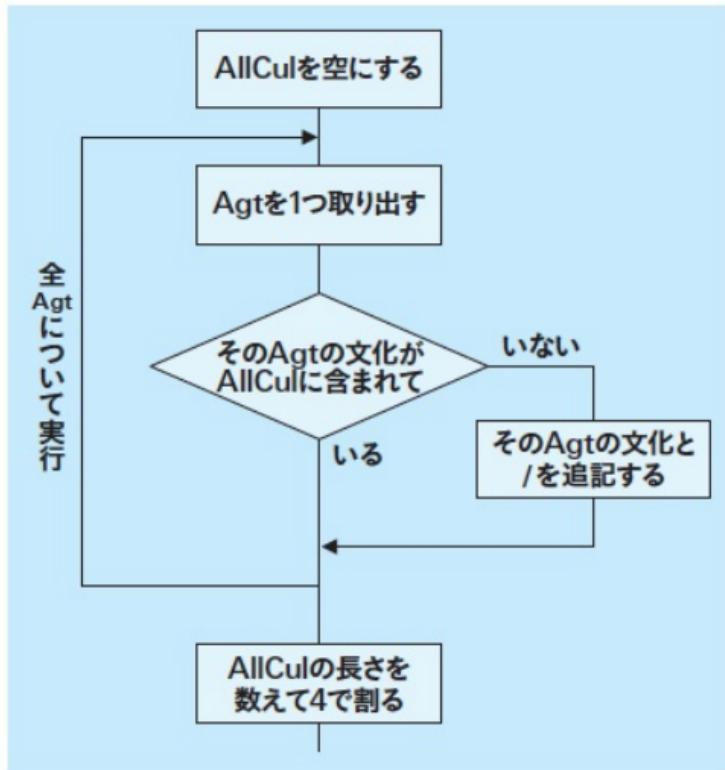


図23.6

次のルール記述でInStr()という関数が登場します。InStr(1, allcul, one.trait)とは、allculという文字列型変数の値になっている文字列の中にone.traitの中の文字列と同じ箇所があるか、allculの1字目からずっと調べるという操作です。作業の結果、もし該当箇所があれば、そこがallculの文字列の何文字目になっているかがInStr(1, allcul, one.trait)の値になります。たとえば10文字目に該当箇所があれば、a = InStr(1, allcul, one.trait)としたとき、aには10が入ります。どこにもなければ0になります。

0なら、今までのレパートリーに含まれていないわけですから、新しいタイプとしてレパートリーに付け加える必要があります。

カラ

なお、文字列型変数の値の初期設定はallcul = ""というふうに表記します。これにより、毎ステップ、空の文字列から作業を始めます。

最後のLen()は文字列の長さを計算する関数です。これを使っている部分では、文字列の長さから文化パターンの数を割り出しています。ちなみに全角文字は1つで2字と数えられます。

では、ルールを見ていきましょう。

```
Univ_Step_End{
Dim total As Agtset
Dim one As Agt
Dim allcul As String
allcul = "" //文字列型変数の初期化
MakeAgtset(total, Universe.country.community)
For each one in total
  If InStr(1, allcul, one.trait) == 0 Then
    allcul = allcul & one.trait & "/"
  End if
Next one
Universe.numcul = Len(allcul) / 4
}
```

上のルールで、文化の多様性を増やす部分

```
allcul = allcul & one.trait & "/"
```

に注目して下さい。最後にスラッシュを追加しています。これは何をしているのでしょうか？上の例では、各エージェントの属性は3文字からなっています。たとえば、3タイプの文化012, 202, 110のレパートリーを表す012202110という文字列があるとしましょう。122や211という文化はレパートリーに含まれていません。しかし2文字目からと6文字目からの3文字分が122と211です。つまり、本来なら文化のレパートリーにまだ含まれておらず、新しく付け加えるべき文化もすでに存在していると判断されてしまいます。したがって、誤読しないようにするために、切れ目を入れておく必要があります。そこでスラッシュを入れると、012/202/110となり、122や211という文化はこのレパートリーに含まれていないと判断されます。このように、スラッシュは文字列の切れ目の役割を果たしています。文化が1つあたり文化の内容を表す3文字とスラッシュ1文字が追加されますから、最後にできあがった文字列の長さを4で割れば、文化の数がわかるわけです。

それでは、文化パターンの数を時系列グラフに出力するように設定して下さい。設定の仕方の説明は省略します。

それではdissemination(D)という名前を付けて保存してから実行してみましょう。文化の多様性の変化をじっくり観察して下さい。

新しく学んだ事項

- 空間の大きさを変数として表す関数GetHeightSpace(), GetWidthSpace()
- ある文字列が文字列型変数の中に存在しているか調べる関数InStr()
- 文字列の長さを調べる関数Len()
- エージェントの変数の値をマップ上に表示する

練習問題

文字列の操作に慣れましょう。



エージェントの行動ルール(23.3を参照)を、なるべくMid()を使わずに、Left()やRight()を使って書き換えてみよう。



工夫すると全然使わないことも可能ですが、そこまでする必要はありません。

23.2

dissemination(B)以下のモデルでは、自分の文化は他のエージェントとの接觸によってでしか変わらない。そこで、次のようなルールを、traitをviewに変換するルール(つまりAgt_Step{ }の最後)の直前に追加してみよう。

每ステップ5%の確率で、どれかの属性のどれかの値がランダムに変化する。



Mid()とCStr()との組み合わせ

第12章で学んだように、エージェントの表示色を変数で指定する場合、その変数型は整数型になります。これは何を意味しているのでしょうか。

実は、artisocでは、各色に、対応する固有の整数値が割り当てられていて、この値を参照することでエージェントの色を表示しています。具体的には、色には数値0から数値16777215までの整数値が割り振られています。0は黒に相当し、16777215は白に相当します。つまり、artisocでは、黒から白まで16,777,216色の色が表示できることになります。

第12章で学んだ色定数(Color_Cyanなど)による配色も、またRGB関数による配色も、artisocがシミュレーション実行時に対応する整数値に置き換えて処理しています。たとえば「My.color = Color_Cyan」というルールを実行すると、My.colorには水色に対応する整数65535が代入されることになります(PrintLn(Color_Cyan)でコンソール出力することで確かめることができます)。また、RGB関数は、以下のような式で赤、緑、青の成分 r , g , b (いずれも0~255の整数値)と色の値とを対応付けています。

$$\text{RGB}(r, g, b) = 256^2 \times r + 256 \times g + b$$

各色が具体的にどのような数値を取るのかを知っておく必要は全くありませんが、エージェントの色が、以上のように、一定の範囲の中の整数値に対応付けられているということを知っていれば、モデルを作る際に何かと便利です。たとえば、エージェントにランダムな色を割り当てるルールも、「My.color = Rnd() * Color_White」という具合に、簡単に書けてしまいます。

(阪)

第24章

エージェントを複雑な条件で選別する

24.0 まとめた上で考えよう

- エージェント集合型変数どうしの操作を学びます
- 集合の操作(演算)は、普通の数の操作(四則演算)とは違います
- 特定のエージェントにも注目します
- エージェントどうしを線で結んで、関係を視覚化します

24.1 エージェント集合の演算に慣れよう

マルチエージェント・シミュレーションでは、エージェントどうしに相互作用させる際に、ある条件を満たした一部のエージェントを対象に特定の行動ルールを適用することが必要な場合が生じます。つまり、エージェントをまとめたり選別したりして、さまざまな「エージェントの集まり」を思いどおりに作れることが重要になります。

「エージェントの集まり」はここで初めて登場する概念ではありません。既に、エージェント集合型変数や、その操作を私たちは頻繁に使ってきました。たとえば `MakeAllAgtsetAroundOwn(neighbor, 2, False)` は、`neighbor`というエージェント集合型変数の中に、自分を中心にして視野2の範囲に存在しているエージェントを全てリストアップする操作です。つまり、特定の条件を満たした「エージェントの集まり」を作っているのです。そして、`GetAgt(neighbor, Int(Rnd() * CountAgtset(neighbor)))`は、`neighbor`にリストアップされているエージェントの集まりから、ランダムにエージェントを1つ選び出す操作です。

では、たとえば「自分との距離が2よりも遠く4以内のエージェント」をどうやって選び出せばよいのでしょうか。`MakeAllAgtsetAroundOwn(neighbor, 4, False)`は視野4の範囲のエージェントを選び出すので、そこから`MakeAllAgtsetAroundOwn(neighbor, 2, False)`で選び出されたエージェントを差し引けば、答え

を得ることができるはずです。どうやれば、前者から後者を「差し引く」ことができるのでしょうか。

一般的な問題として考えれば、エージェントを選び出す基準は視野(自分との距離)だけとは限りません。さまざまな条件で、その条件を満たした(あるいは満たしていない)エージェントを選び出す必要があります。

この章では、「エージェントの集まり」を利用して特定の条件に当てはまるエージェントを選別するさまざまな技法を学びます。エージェント型変数やエージェント集合型変数を対象にした演算に慣れましょう。

24.2 友達の輪

さまざまな条件に基づくエージェントの選別とは、「エージェントの集まり」の演算(操作)をすることです。モデルを作る過程で、このような考え方慣れるようにしましょう。この章では、「エージェントの集まり」として、仲間ができるいく過程をモデル化しましょう。

社会に音楽と漫画についてさまざまな好みを持った人々がいる。人は、好みの近いものを友人にする。

では、いつものように準備作業に取りかかりましょう。

1. Universeの下に空間society(「ループしない」を選択し、それ以外はデフォルトで)を作る。この空間は、横軸がひとつの好み、縦軸がもうひとつの好みを表し、それぞれの好みは0から50までのスケールで計るものとしましょう(これで、なぜ空間をループさせないのか、わかりますね)。
2. societyにはperson(エージェント数200)を作ります。personのX, Yが個々のエージェントの2種類の関心についての好みを表すものと考えましょう。たとえばバロックからハード・ロックまでがX軸に並び、ストーリー劇画からナンセンス4コマまでがY軸に並んでいる、などと想定して下さい。personには友達(「友人の集まり」)を表すエージェント集合型変数friendsを追加しましょう。
3. マップ出力も設定しておきます。X, Yの意味をこのように考えることにより、マップ上のエージェントの位置はそのエージェントの好みを表すことになります。

4. これをfriendshipという名前で保存しましょう。

このモデル作りでは、Universeのルールは特に使いません。以下では、さまざまなエージェント集合型変数を扱う操作技法を少しずつ学んでいきます。

まず、エージェントを初期設定します。

```
Agt_Init{  
    My.X = Rnd() * 50  
    My.Y = Rnd() * 50  
    My.Direction = Rnd() * 360  
    ClearAgtset(My.friends) //friendsを初期化(中を空にする)  
}
```

ClearAgtset()は新しいルール表現です。この関数は、括弧の中のエージェント集合型変数を初期化(中を空に)します。friendsはツリーの中の(エージェントに追加された)エージェント集合型変数なので、自動的に初期化されますが、念のため、明示的なルールで初期化しましょう。初期化されていないエージェント集合型変数を参照するルールを書くと、実行時にエラーが生じことがあります。

24.3 友人を作ろう

好みの近い人どうしが友達になる過程をモデル化します。なお、ここでモデル化する友達関係は、一方指向的です。つまり、PにとってQが友人だとしても、QにとってPが友人だとは限りません。

好みの違い(視野)が3以内のエージェントをひとつ選んで友人にする(自分の友達に加える)。友人の好みに近づく。

```
Agt_Step{  
    Dim s As Integer  
    Dim one As Agt  
    Dim temp As Agtset  
    Dim close As Agtset //次節で使います  
    Dim neighbor As Agtset  
    Turn(Rnd() * 360) //とりあえず、好みはランダムに変わろうとする  
    //自分と好みの似ている人を友人にする。
```

```

MakeAllAgtsetAroundOwn(neighbor, 3, False)
If CountAgtset(neighbor) > 0 then
    one = GetAgt(neighbor, Int(Rnd() * CountAgtset(neighbor)))
    AddAgt(My.friends, one) //one をMy.friendsに追加
    TurnAgt(one) //oneの好みに合わせようとする
    //My.friendsの重複をなくす
    DuplicateAgtset(temp, My.friends)
    PurifyAgtset(My.friends, temp)
End if
Forward(1)
}

```

新しい表現がいくつか登場したので説明しておきましょう。

AddAgt(My.friends, one)(第21章で登場)により、周囲(視野3の範囲)にエージェントがいればランダムに選んで、それ(one)を自分の友達(My.friends「友人の集まり」)に加えます。

これで十分のようですが、実は違います。後になって既に友人にしたエージェントを新しい友人として再び選別してしまう可能性があります。そこで、重複をなくすために

```

DuplicateAgtset(temp, My.friends)
PurifyAgtset(My.friends, temp)

```

という2つの新しい関数が続いています。まず、DuplicateAgtset()はMy.friendsをtempに複製します。PurifyAgtset()はtempの重複を取り除いてMy.friendsにします。これで、My.friendsの中には重複のない友達がいます。こうした上でCountAgtset(My.friends)を使うと、正確な友人数が求まります。oneを友達に加える前に、既に友達に含まれているかどうか調べる方法もありますが、調べる作業をartisocに任せてしまう上のルールの方が簡単です。なお、この2行のルールをまとめて、PurifyAgtset(My.friends, My.friends)と書くことも可能です。

それでは、上書き保存してから実行しましょう。

上の行動ルールでは、近くにいる人のうち、友人に対するのはどれか一人です。全員まとめて友人に対するルールにするには、既存の友人たち(My.friends)に近くにいる人(neighbor)を加える操作をすることになります。これは、組み込み関数MergeAgtset()を使って

```
MergeAgtset(My.friends, neighbor)
```

と表記します。この関数は、自動的に重複を除外しますから、PurifyAgtset()の操作が不要です。

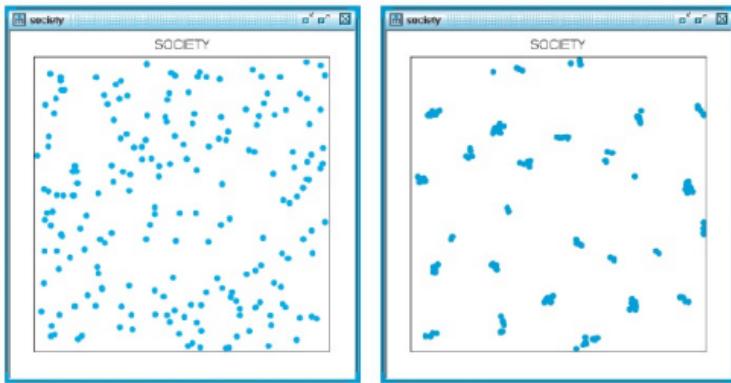


図24.1

24.4 友達になれば不即不離

上の行動ルールでは、似たもの同士が集まり、小さな集団がたくさんできました。そこで、もう少し複雑にしてみましょう。

いったん友達どうしになると、お互いの好みが離れると近づけようとするが、あまり似すぎるとかえって好みを変えようとする。

具体的には、次のようなルールにしましょう。

1. 好みが3より離れてしまった友人がいれば、その人に近づく。
2. 好みが1.5より近づいてしまった友人がいれば、その人から遠ざかる。

上のルールの最後の「Forward(!)」の直前に、下のルールを書き込みます。このルールは、行動1が行動2に優先しています。

```
//離れた友人の好みに合わせようとする
DuplicateAgtset(temp, My.friends) //友達全員のコピー
DelAgtset(temp, neighbor) //好みの近い人を除く
If CountAgtset(temp) > 0 Then //好みの離れた友人がいれば
    one = GetAgt(temp, Int(Rnd() * CountAgtset(temp)))
    TurnAgt(one) //友人の好みに合わせようとする
Else
    //好みの近すぎる友人から離れようとする
    MakeAllAgtsetAroundOwn(close, 1.5, False) //好みが近すぎる人々
    MakeCommonAgtset(temp, close, My.friends) //好みが近すぎる友達
    If CountAgtset(temp) > 0 Then //好みが近すぎる友人がいれば
        one = GetAgt(temp, Int(Rnd() * CountAgtset(temp)))
        TurnAgt(one)
        Turn(180) //結局、反対方向を向く
    End if
End if
Forward(1)
}
```

新しい技法が登場しました。

```
DelAgtset(temp, neighbor)
```

先ほど調べた周囲のエージェント集合neighborをtemp(友達)から取り除きます。これによりtempには好みの離れた友達が残ります。この際、DelAgtset(My.friends, neighbor)と表記してしまうと、好みの近い友人たちが友達でなくなってしまいます。そのために、DuplicateAgtset()で複製したtempを操作対象にしているのです。なお、neighborにはtempに含まれない(友人ではない)エージェントも混じっている可能性がありますが、tempへの操作に影響を与えないで無視してかまいません。

```
MakeCommonAgtset(temp, close, My.friends)
```

近すぎるエージェント(必ずしも全員友人とは限りません)closeと友達(友人全員)My.friendsとに共通する者をtempとする操作です。つまり、二つの「集まり」の共通部分をtemp という「エージェントの集まり」にするのです。なお、

```
MakeAllAgtsetAroundOwn(close, 1.5, False)
MakeCommonAgtset(temp, close, My.friends)
```

のcloseをtempにしてもかまいません。MakeAllAgtsetAroundOwn()で求めたtempとMy.friendsとの共通部分を、MakeCommonAgtset()の中のtempにしまうので、支障がないのです。わたしたちの頭の混乱やルールの混乱が起こらなければ、変数の「使い回し」はかまいません。

friendship(B)という名前を付けて保存してください。このモデルを実行すると、ばらばらに散らばっていたエージェントが揺れ動きながら、だんだんと固まっていく様子が観察されます。次第に、大小いくつかのクラスターにまとまっていくでしょう。前節より大きなまとまりができたのではないですか。

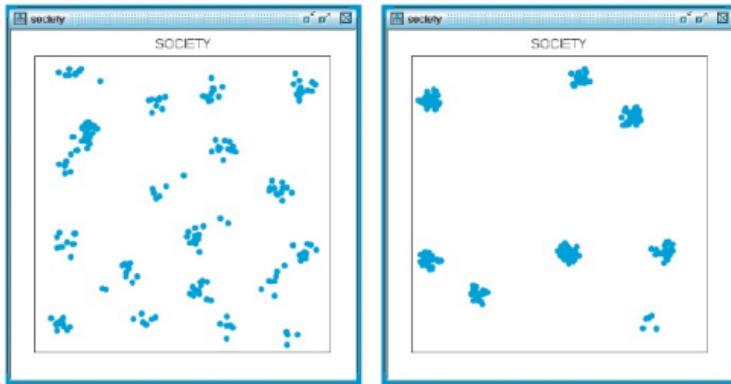


図24.2

24.5 友人を線で結んで明示的に示す

ところで、誰と誰とが友達なのでしょうか？ 友人どうしは一定程度好みが似ている（マップ上で近くにまとまっている）から、何となく想像はできますが、はっきりとはわかりません。友人関係をマップ上に表すことができれば、便利です。

それでは、自分と自分の友人を線で結ぶことにしましょう。一般的に言うと、各エージェントとそのエー

ジェントのエージェント集合型変数に入っているエージェントとを、マップ上の線で結ぶ技法です。このモデルでは、friendsには友人として選ばれたエージェントがリスト化されています。したがって、各エージェントについて、自分のfriendsの中のエージェントと線で結ばれるのです。

どうすればよいでしょうか。

これはマップ出力で設定します。すでにマップの出力は設定されていますから、編集作業をします。

要素設定ウインドウで、「線を引く」にチェックを入れると、線を結ぶために必要ないくつかの設定ができるようになります。線引対象では、friendsを選びます(personの下のエージェント集合型変数はこれしかありませんから、他の選択肢はないはずです)。たとえば、[図24.3](#)のようにしてみましょう。

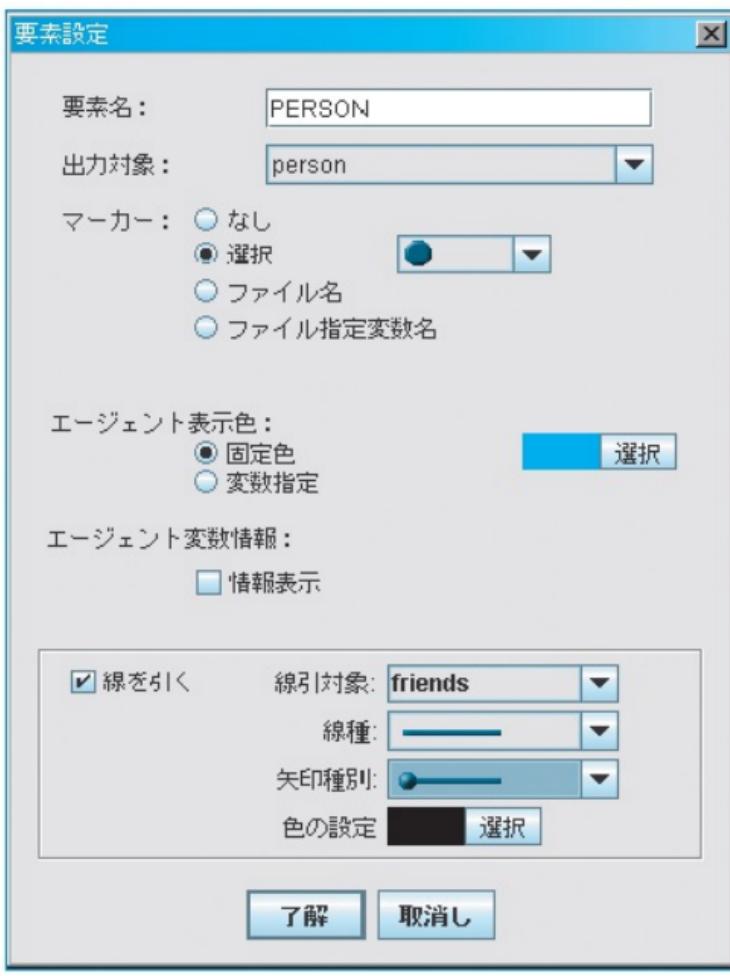


図24.3

これで完了。簡単でしょう。了解ボタンを順次押して、出力設定を終わらせます。それでは friendship(C)という名前を付けて保存してから実行して、友人どうしを線で結ばなかったときと、このように

線で結ぶ出力設定をした場合とを比較しましょう(図24.4参照)。

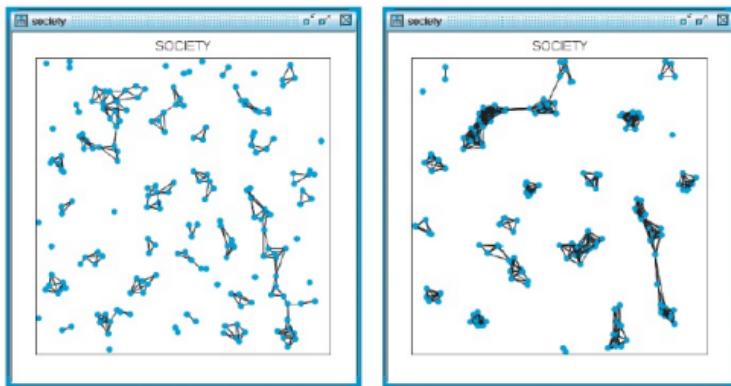


図24.4

エージェントとエージェントとを線で結ぶというのは、このモデルでは必ずしも本質的部分ではありません。しかしシミュレーションの過程を観察するときに、威力を発揮します。これからは、エージェントとエージェントの特別な関係を目の前に示してみなければ、エージェント間に線を引くという技法を駆使して試みて下さい。次章では、この技法を応用して、複雑な関係を表す方法を学びます。

新しく学んだ事項

- 「エージェントの集まり」を操作する技法
- エージェント集合型変数を操作する関数ClearAgtset(), DuplicateAgtset(), DelAgtset(), PurifyAgtset(), MergeAgtset(), MakeCommonAgtset()
- マップ上で、特定のエージェントどうしの間にリンクをはる

練習問題

友人関係を線で表してみると、かなり離れたエージェントどうしも友達の場合があることがわかります。

そこで、好みが離れすぎた場合には友人であることをやめる、という行動ルールを追加してみましょう。

視野が8以上離れた友人がいれば、友達から除外する。

24.1

複数いる場合は、全員を除外するようにしてみよう。



DelAgtset()をうまく使って、遠くにいる友達の集合を作ります。

24.2

複数いる場合は、ランダムで一人だけ除外するようにしてみよう。



第3、4節で学んだ操作方法を組み合わせて使えばルールを作ることが可能ですが、AddAgt(agtset, agt)の逆、つまり特定のエージェントを除く組み込み関数RemoveAgt(agtset, agt)を使うと便利です。

第25章

エージェントを識別して複雑な関係にする

25.0 相手が違えば、関係も変わります

- 個々のエージェントを識別することが可能です
- 相手ごとに関係を区別します
- ID(アイディー)と配列とを組み合わせる技法を学びます
- クラスマートの相互作用のモデルを作ります

25.1 顔見知り社会の複雑な関係に注目する

エージェントどうしの相互関係がマルチエージェント・シミュレーションの基本であることは何回となく強調してきました。今までには、たまたま周囲にいたエージェントとの相互作用が中心でした。それに対して、特定のエージェントとの相互関係をモデル化することが必要な場合もあるでしょう。この章では、特定のエージェント間の関係をモデル化する技法を学びます。

エージェントの属性を複雑にする場合とエージェント間の関係を複雑にする場合とでは、複雑度のレベルが異なります。エージェントの属性がいかに複雑だろうと、個々のエージェントについて属性の内容が決まっています。これに対して、関係を複雑化させるときには、個々のエージェントについて他のエージェントとの関係をはっきりさせる必要があります。そのためには、エージェントの「個体識別」と、各エージェントが他のエージェントとどのように異なる関係になっているかを表す「配列」をルールの中で操作することが基本になります。

これから学ぶ技法を利用することによって、個体識別に基づいた複雑な関係のモデル化が可能になります。これにより、顔見知り社会における個人間の複雑な関係を扱えます。

25.2 個体識別はIDで

モデル作成の過程で空間にエージェント(種)を追加すると、必ず、ID, X, Y, Layer, Directionという変数が自動的に作られます。今までではX, Y, Directionのみをルールで使ってきましたが、この章ではいよいよIDをルールの中で使います。

初期設定だろうと、実行中だろうと、エージェントが生成されると、エージェント種ごとに、生成された順に0から順次、整数をIDに割り当てていきます。一度割り当てられたらシミュレーションが終了するまで変わりません。このように個々のエージェントに与えられる個別の番号をID(アイディー:Identityアイデンティティの略)と呼びます。DelAgt()を実行して、エージェントを消滅させても、番号がずれたりしません。このIDを利用して、モデルの中でエージェントを個体識別するというのが出発点です。

IDは個体識別のための特別な変数です。わたしたちがIDの値を設定したり変更したりできません。したがって、

```
My.ID = 5
```

などと代入しようとするとエラーになります。しかし

```
i = My.ID
```

のようにIDの値を何かに代入することはできます。もちろん、artisocが自動的に割り当てるエージェントIDではなく、自分でツリーのエージェント種に、たとえばIDNoという名前の変数を追加して、エージェント毎に異なる固有な値を与えて個体識別に用いることも可能です。

25.3 多数の変数を配列にまとめる

エージェントが固有の番号を持っていることを前提にすると、個々のエージェントを区別して、エージェント毎の複雑な関係を表現できます。学校の同級生の相互関係を例にとると、40人のクラスでは、個々の生徒は自分以外の39人との付き合いを区別しているでしょう。

このようなとき、各生徒の関係を表す指標をID順に40個並べておいて、自分のIDのところは空欄にしておけば便利です。このようなことを、artisocでは、1つの変数を多数の要素を持つ配列にすることで実現できます。つまり40個の変数の代わりに、1つの変数（たとえばrelation）に40の値を配列するわけです。これを、配列数40の1次元変数といいます。生徒どうしの関係を、仲良く遊ぶ関係とけんかする関係とに分けて表す場合、たとえばplayとfightという2つの1次元配列変数で表すこともできますが、relationを2次元配列変数に設定し、2次元目の配列数を2にして、遊びとけんかの両方をひとつの変数にまとめることも可能です。

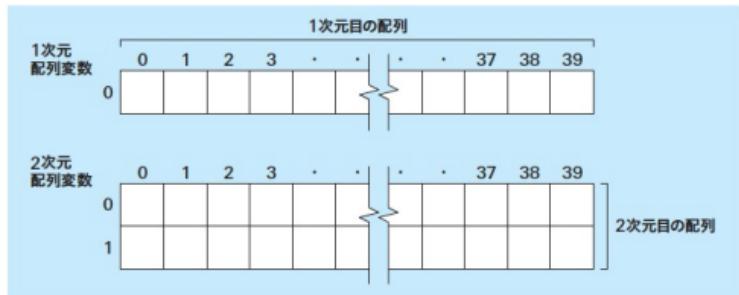


図25.1

配列のある変数を作るには、artisocの変数のプロパティ・ウィンドウで「次元数」と各次元の「配列数」を指定します。artisocでは、ID、X、Yなどツリーに自動的に作られる変数を除き全ての変数を多次元に設定し、各次元を多数の配列に設定することができます。ただし空間変数だけは、空間の構造(X、Yの2次元とレイヤ数)を反映するので、常に3次元の配列となっています。その意味では、「配列」という名前こそいませんでしたが、第18章で学んだ空間変数を扱う技法は、配列を操作する技法だったので。なお、空間変数が配列になっていたことからも明らかなように、配列はIDと関連づける必要は必ずしもありません。

25.4 同級生の関係をモデル化する

それでは、同級生どうしの関係が変化するモデルを作ってみましょう。

教室の中で、近くにいる同級生どうしが遊んだりけんかしたりする。遊ぶことの多い同級生を好きになり、けんかすることの多い同級生を嫌いになる。

学級の中では互いに相手のことを知って付き合っているので、各人は他の同級生のひとりひとりについて好悪の程度や交友関係の粗密を持っているはずです。その変化をモデル化します。

1. Universeの下にclassroom(空間の大きさを 20×20 に設定し、「ループしない」を選択します)、その下にpupil(エージェント数40)を作ります。
2. pupilにrelationという「配列変数」を実数型で作り、次元数を2にし、1次元目の配列数を40、2次元目の配列数を2に設定しましょう([図25.2](#)参照)。

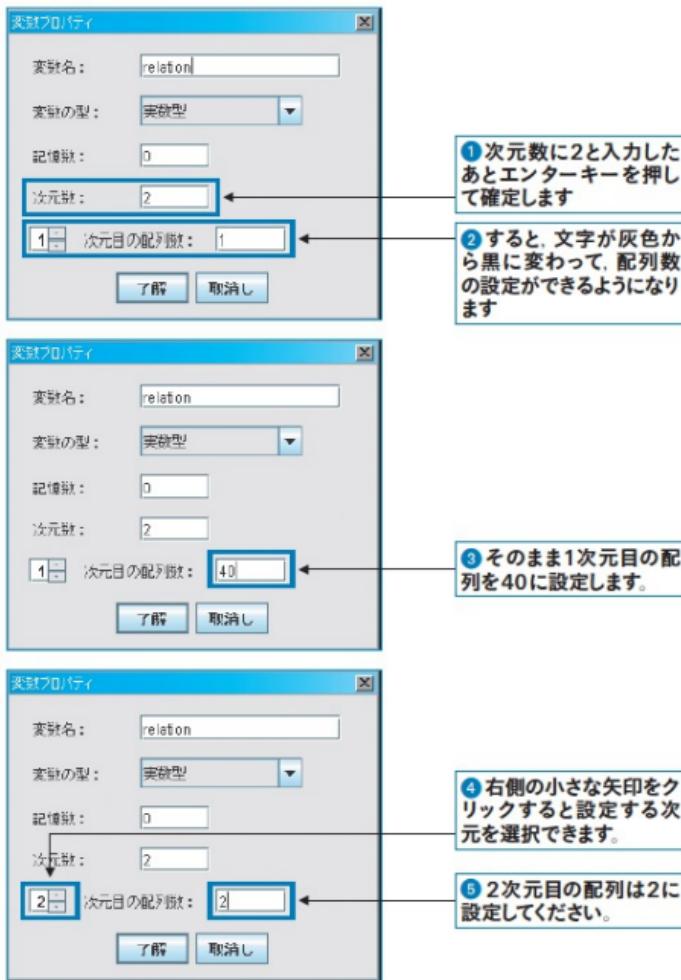


図25.2

こうして作った変数relationは、relation(i, j)と表し、iは0から39までの値を、jは0か1の値をとりま

す。各エージェントにとって、relation(i, j)には、IDがi(i番目)のエージェントとの遊んだ度合い(relation(i, 0)の中)とけんかをした度合い(relation(i, 1)の中)がしまわれます。pupilにはさらにエージェント集合型変数friends, enemiesを追加します。

3. マップ出力の設定をして下さい。pupilの要素設定の際、「線を引く」をチェックし、friendsを線引対象にして下さい。このとき、[図25.3](#)のように、実線で矢印付きの線を選択しましょう。

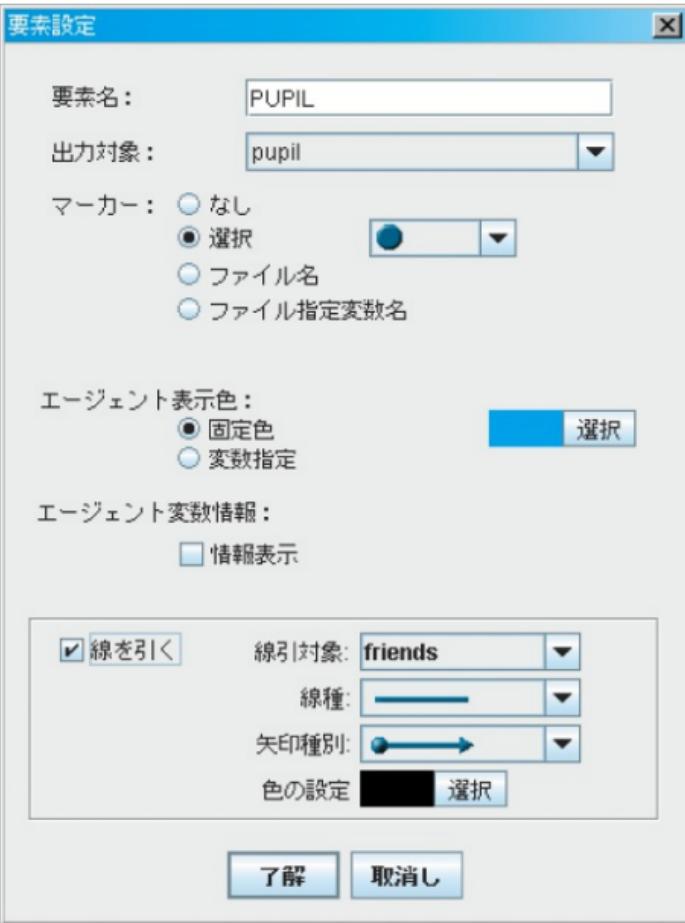


図25.3

4. モデルをclassmatesと名付けて、保存して下さい。

具体的には、次のようなルールにしましょう。

- 生徒は教室の中をぶらぶら歩き回り、近くにいる生徒と遊ぶか、けんかする
- 遊ぶ(けんかする)経験がけんかする(遊ぶ)経験を凌駕(差が3以上になる)すると、その相手を好きな(嫌いな)相手とする

まず初期状態を与えます。

```
Agt_Init{
My.X = Rnd() * 20
My.Y = Rnd() * 20
My.Direction = Rnd() * 360
}
```

エージェントの行動ルールは次のようになります。空間がループしていないので、動き回るルールに第21章で学んだ工夫が必要です。

```
Agt_Step{
Dim i As Integer
Dim one As Agt
Dim neighbor As Agtset
Turn(Rnd() * 180 - 90)
If Forward(1) <> -1 Then //前進してぶつかったらUターン
  Turn(180)
End if
//近くにいる同級生と遊ぶかけんかする
MakeAllAgtsetAroundOwn(neighbor, 2, False)
For each one in neighbor
  i = Round(Rnd()) //四捨五入して0(遊ぶ)か1(けんか)
  My.relation(one.ID, i) = My.relation(one.ID, i) + 1 //関係を蓄積
Next one
//同級生と恒久的関係の構築
ClearAgtset(My.friends) //好きな相手を初期化
ClearAgtset(My.enemies) //嫌いな相手を初期化
MakeAgtsetSpace(neighbor, Universe.classroom)
For each one in neighbor
  If My.relation(one.ID, 0) >= My.relation(one.ID, 1) + 3 Then
    AddAgt(My.friends, one) //好きな相手に加える
  Elseif My.relation(one.ID, 1) >= My.relation(one.ID, 0) + 3
  Then
    AddAgt(My.enemies, one) //嫌いな相手に加える
  End if
Next one
```

}

ここでRound(Rnd())というルール表記が登場しました。Round()は小数点以下を四捨五入する関数です。したがって0以上1未満の乱数は0か1になります。上書き保存してから、実行してみましょう(図25.4参照)。

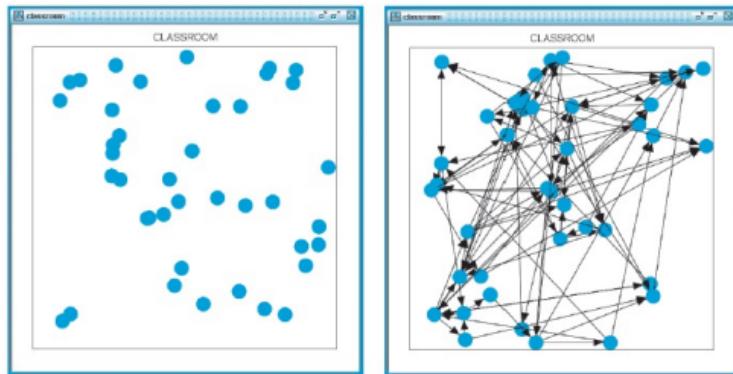


図25.4

25.5 複数の関係を視覚化する

上で作ったモデルでは、好きな相手だけでなく嫌いな相手もできたはずですが、マップ上には好きな相手しか図示されていません。それは、1つのマップ要素からは1種類のエージェント集合型変数の中身(個々のエージェント)と1種類の線しか引くことができないからです。

どうしたら、嫌いな相手もマップに図示することができるでしょうか。

それは、再度pupilをマップ出力要素に追加しておいて、こちらの出力要素の設定で、今度はenemiesを線引対象に選択すればよいのです。そうすれば、異なった線が引かれたpupilが重なって表示されます。この際、線の種類や色をfriendsの線と区別できるように設定して下さい。

このように出力設定を変えたモデルを、classmates(B)という名前で保存して実行して下さい(図25.5参照)。

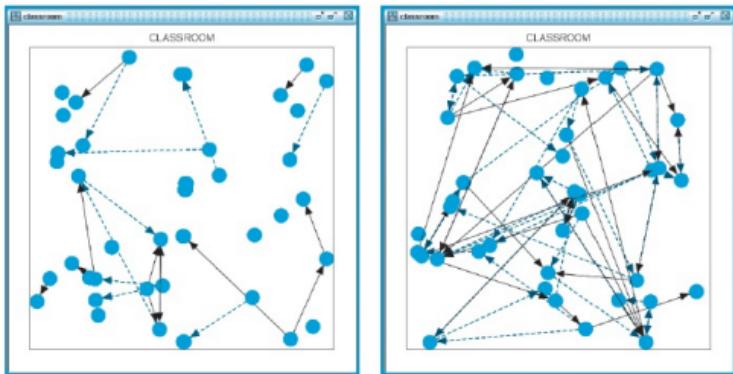


図25.5

2種類の矢印がマップ上に出てくるでしょうか。第18章でも触れましたが、マップ出力は出力要素リストの順に重なって表示されます。エージェントの色(固定色)や線は、見栄えがよいように、適宜選択して下さい。

25.6 配列の扱いに慣れる

classmates(B)モデルのままでは、生徒どうしの関係はどんどん累積されるだけです。また、関係の累積は教室を歩き回る行動になんら影響を与えません。そこで、次のようなルールを追加することにしましょう。

関係は毎ステップ10%ずつ減っていく一方、関係が増えるほど、うろうろしなくなり、交友範囲は広くなる。

そこで、pupilに実数型変数sociabilityを追加して、次のルール変更をします。sociabilityは、生徒の社交性を表す指標で、2と6の範囲の値をとるものとしましょう。

Agt_Init{ }の最後に追加します。

```
My.sociability = 2
```

Agt_Step{ }の中のルール2カ所を変更します。

1. If Forward(1) <> -1 Then を次のように変更

```
If Forward(2 / My.sociability) <> -1 Then
```

2. MakeAllAgtsetAroundOwn(neighbor, 2, False)を次のように変更

```
MakeAllAgtsetAroundOwn(neighbor, My.sociability * 2, False)
```

さらにAgt_Step{ }に、次のルールを追加します。ここで、入れ子構造のフォ文だけでなく、配列変数relationの扱い方に注意して下さい。

```
Dim j As Integer //冒頭に追加  
//最後に追加  
My.sociability = 0 //初期化  
For i = 0 To 39  
    For j = 0 To 1  
        My.sociability = My.sociability + My.relation(i, j)  
        My.relation(i, j) = My.relation(i, j) * 0.9  
    Next j  
Next i  
My.socialbility = My.sociability / 39  
If My.sociability > 6 Then  
    My.sociability = 6  
Elseif My.sociability < 2 Then  
    My.socialbility = 2  
End if
```

以上で修正は完了です。classmates(C)という名前で保存して実行して下さい。生徒の行動や生徒どうしの関係に変化が生じたのを、見て取ることができるでしょうか。

新しく学んだ事項

- エージェントの個体識別
- エージェント変数ID
- 配列のある変数の設定と扱い方
- マップに複数種類の線を引く方法
- Round()

練習問題

classmates(B)モデルでは生徒が動き回っていましたが、動かないルールにして、教室にいる誰とでも遊んだり、けんかをしたりするモデルに修正しましょう。恒久的関係の構築や、マップ上で関係を線で表示する部分はそのまま活かします。

25.1

毎ステップ、ランダムに選ばれた一人の生徒と、遊びかけんかする。



IDを利用します。もし自分自身が選ばれたら、何もしないルールにしましょう。

25.2

練習問題25.1のルールに加えて

相手のrelationにも自分との関係(遊びかけんか)を蓄積する。



フォ・イーチ文で相手をさがして、見つかったら作業をしてからBreak。

Column

フォ・イーチ文とRemoveAgt()の使用上の注意

artisocでは、一群のエージェント全てと相互作用するにはフォ・イーチ文を、一つだけ選び出したエージェントと相互作用するにはGetAgt()関数を用います。エージェント同士の相互作用に関わるこの二つの技法は、artisocでマルチエージェント・シミュレーションのモデルを作る際に頻繁に用いられます。両技法を使用する際の注意点を以下にまとめておきましょう。

第13章で学んだように、フォ・イーチ文では「For Each a In S XXXX Next a」の形式で、指定されたエージェント集合型変数Sの要素a(エージェント型変数)を順々に巡って、ルールXXXXを繰り返し実行していきます。その際、ルールXXXXの中で、個別のエージェントの状態や属性を操作することはできますが、集合Sからエージェントを削除したり、逆に追加したりするなど、エージェント集合Sの構成そのものを変えようとすると、エラーが生じてしまいます。たとえば、周囲のエージェントからある条件を満たすエージェントだけを選別しようとする場合、以下のようなルールを書いてしまいがちです。

```
For Each hito In neighbor
    If hito.wealth <= 1000 Then
        RemoveAgt(neighbo, hito)
    End If
Next hito
```

この場合、ルールの繰り返しの過程でRemoveAgt()関数を使ってエージェント集合型変数neighborの構成を変えてしまうので、エラーが発生することになります。これを回避するには、以下のように、フォ・イーチ文で用いる集合とRemoveAgt()などの操作の対象となる集合とを別個のものにする必要があります。

```
DuplicateAgtSet(rich, neighbor)
For Each hito In neighbor
    If hito.wealth <= 1000 Then
        RemoveAgt(rich, hito)
    End If
Next hito
DuplicateAgtSet(neighbor, rich)
```

(版)

第26章

繰り返しの多いルールをすっきりと：レイノルズのボイド・モデル（2次元版）を作る

26.0 自分仕様のartisocにしよう

- 自分専用の関数（ユーザ定義関数）を作れます
- 繰り返しの多いルールをまとめると便利です
- ユーザ定義関数には2つのタイプがあります
- 両方ともここで学びます
- ボイド・モデルの2次元版を作りましょう
- ユーザ定義関数を別々のモデルで使うことも簡単にできます

26.1 ルールをまとめるメリット

今までにさまざまなタイプのモデルを作ってきました。その時、同じようなルールを何回も書いた経験があったでしょう。そのような場合、同じ類のルールを1回書き込んでおいて、それを使い回しできると便利です。この章では、まさに、この便利さを追求します。つまり、繰り返しの多いルールをひとつにまとめる技法と、それをルールエディタの中で何回も使う技法を学びます。

そのヒントは関数です。モデルを作る際、既に私たちはエージェントのルールの記述で、たとえばTurn()やMakeOneAgtsetAroundOwn()を使ってきました。これらは全て、何か特定の作業をする関数です。既にartisocの中に組み込まれている関数なので、「組み込み関数」と呼びます。組み込み関数は多数ありますが、その多くは何行にもわたるルールをひとつの名前にまとめたものです。artisocでは、モデルを作るユーザが自分専用の関数を作り、ルールを記述するときにそれを使うことができます。artisocのユーザが自分で勝手に定義して使うので、組み込み関数に対して、「ユーザ定義関数」と呼びます。

この章は、ユーザ定義関数の作り方と使い方がテーマです。

26.2 ボイド・モデルは簡単に作れる

ユーザ定義関数がどんなに便利なのかを実感してもらうために、とても有名なモデルを作ります。

人工生命としてまとめられているさまざまなモデルがありますが、そのうちの一つ、ライフゲームは既に第20章で作りました。別な有名なモデルに、クレイグ・レイノルズが考案したボイド・モデルがあります。鳥(bird)に似たもの(-oid)を短くしたボイド(boid)です。このモデルは、少數の簡単なルールを与えると、ボイドと名付けられたエージェントは本物の鳥のように群れを作り、障害物をよける、というものです。オリジナルのモデルは3次元モデルですが、ここではartisocの空間にあわせて2次元のモデルを作りましょう。つまり、左右には曲がるが上下には変化しないで飛ぶ鳥のモデルです。

最初に、ボイドに群行動をさせましょう。オリジナルのモデルよりも単純なルールで群ができる過程を再現することが可能です。

準備作業から始めます。

1. Universeに空間country(設定はデフォルト)を追加し、そこにエージェントbirdをエージェント数20として追加する
2. birdに飛ぶ速さを指定する実数型変数speedを追加する
3. countryをマップ出力するように設定する。birdを要素として追加する(birdのマーカーを▲の形で表すと見かけがきれいです)
4. モデルをboidという名前を付けて保存する

近くにいるトリと同じ方向に同じ速さで飛ぶようになる。基本はこれだけです。これだけで群ができます。オリジナルなモデルには、群の重心に向かう、といった複雑なルールが含まれているようですが、私たちがこれから作るモデルには不要です。

- 最初は、ランダムな位置・方角・速さにする
- 周り(視野の広さ2)にいるトリを、自分の仲間だと見なす
- 仲間のうちの1羽の方向と速さに自分の方向と速さを合わせる
- 仲間がいなければ、方向や速さを適当に変える

20羽のトリの初期状態は次のようにします。

```
Agt_Init{
My.X = Rnd() * 50
My.Y = Rnd() * 50
My.Direction = Rnd() * 360
My.speed = 0.2 + Rnd() * 0.3
}
```

各ステップのトリの行動ルールは次のようにします。

```
Agt_Step{
Dim neighbor As Agtset
Dim one As Agt
//群を作るルール
MakeOneAgtsetAroundOwn(neighbor, 2, Universe.country.bird,
False)
If CountAgtset(neighbor) > 0 Then //仲間がいれば
  one = GetAgt(neighbor, Int(Rnd() * CountAgtset(neighbor)))
  My.Direction = one.Direction
  My.speed = one.speed
Else //仲間がいなければ
  Turn(Rnd() * 30 - 15)
  My.speed = 0.2 + Rnd() * 0.3
End if
Forward(My.speed) //向いている方向に飛ぶ
}
```

以上がトリ・エージェントの行動ルールです。上書き保存してから実行して下さい。このルールだけで、最初はバラバラに飛んでいるトリ・エージェントが、徐々に群れを作っていく様子が観察できます。

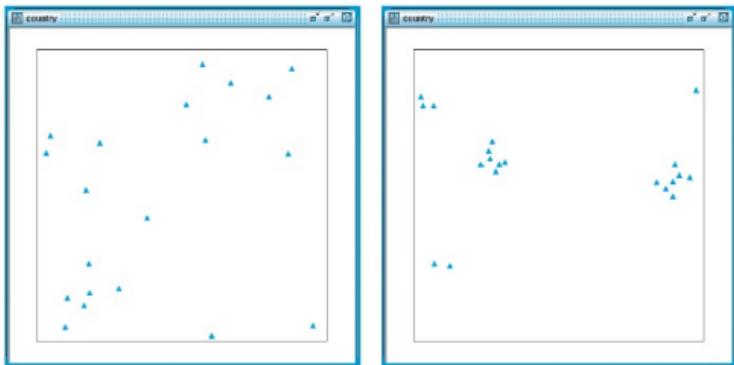


図26.1

以上でボイド・モデルの基本は完了ですが、ボイド・モデルの特徴は、トリ(ボイド)が飛ぶうちに群がひとりでにできるだけでなく、群が障害物を避けながら飛ぶところにあります。モデルを完成させるためには、countryに障害物barrierを置く作業と、birdが前方に障害物がないか調べて、あれば避ける行動ルールを加える作業が必要です。以下では、「ユーザ定義関数」を作ることによって、そうした作業のルール表記をすっきりとさせる技法を学びます。

26.3 ユーザ定義関数を作って、障害物をばらまく

それでは、トリが避けていく障害物を空間にばらまきましょう。country下にbarrier(エージェント数0)を追加して下さい。countryのマップ出力を編集し、barrierをマップ要素に追加して下さい。この際、実行時の見かけがきれいなのでマークー■を選択して下さい。このモデルは、新しくboid(B)という名前を付けて、保存して下さい。

では障害物をランダムに配置するルールを書きましょう。障害物は実行時には何もしないのでエージェントのルールエディタには何も書き込みません。Universeのルールエディタで初期配置するだけです。

障害物エージェントは、出力されるマップの上では(マークー■を選択したので)ブロックのように見えます

が、実際には空間上の点です。birdが避けて飛ぶ様子が見やすくなるように、ある程度大きい障害物にする必要があります。そこで、1つの障害物をランダムに配置したら、その隣に障害物をいくつかまとめて並べるルールにします。

1. countryの端に置くと見づらいので、端を除外して、ランダムに障害物を20配置する
2. その上隣または右隣に1だけ離して障害物を置く
3. 両者の中間にも障害物を置く

上のように障害物を置くとすると、1, 2, 3の全てで、障害物を置くという作業をすることになります。この作業を「ユーザ定義関数」として、ひとつにまとめ、繰り返して使うことにしましょう。

ひとつのまとめた作業をさせる一連のルールは、Sub(「サブ」と読みます。サブルーチンというプログラミング用語の省略形です)というタイプのユーザ定義関数としてまとめることができます。ここでは、障害物を置くSubを作ります。

Univ_Init{ }で障害物を置くという作業は

1. 障害物を生成する
2. 生成された障害物を指定の位置、つまりX座標とY座標の(XX, YY)に置く

というルールです。これをBuildBarrierという名前のユーザ定義関数にしましょう。具体的には、次のように記述します。

```
Sub BuildBarrier(XX As Double, YY As Double){  
Dim obj As Agt  
obj = CreateAgt(Universe.country.barrier)  
obj.X = XX  
obj.Y = YY  
}
```

BuildBarrier(XX As Double, YY As Double)とあるように、ユーザ定義関数の()の中には、作業の際に実際に指定されるXXとYYが実数型変数であると宣言しておきます。中括弧[]を書くのを忘れな

いで下さい。

ユーザ定義関数は、必ず、それを使うルールエディタの最後部に書き込みます。Sub BuildBarrier()は Universe のルールエディタの中で使うので、Univ_Finish{ }の後に Sub BuildBarrier(){ }を書き込みます(図26.2参照)。そしておいて、このユーザ定義関数を Univ_Init{ }のセクションの中で組み込み関数と同じように、BuildBarrier()として使えばよいのです。



図26.2

その際、ユーザ定義関数を定義する部分 Sub BuildBarrier(XX As Double, YY As Double)と、それを Univ_Init{ }の中で BuildBarrier(H, V)として使う箇所とで、括弧の中の変数はきちんと対応(XXは H に、YYは V に対応)している必要があります。要するに、ユーザ定義関数の使い方は、

1. 関数を使ってどのような操作をするのかを明記する(関数の定義)
2. 具体的に、その関数を利用する(関数の呼び出し)

が基本です。今まで使ってきた「組み込み関数」は、既に 1. 関数の定義がなされていて、私たちは 2. それを利用するだけでよかったですのに対し、「ユーザ定義関数」は、1. も自分で行なう必要があるのです。

さて、下記が Sub BuildBarrier() を使って障害物をばらまくルールです。

```

Univ_Init{
Dim i As Integer
Dim h As Double //X座標
Dim v As Double //Y座標
Dim n As Double
For i = 0 To 19
    h = Int(Rnd() * 40) + 5 //5以上44以下
    v = Int(Rnd() * 40) + 5
    n = Round(Rnd()) //四捨五入して0か1
    BuildBarrier(h, v)
    BuildBarrier(h + n, v + 1 - n) //上隣か右隣
    BuildBarrier(h + n / 2, v + (1 - n) / 2) //両者の中間
Next i
}

```

20の障害物を端の方には置かないようにランダムに配置し、その右隣か上隣にもう1つ障害物を追加し、さらにその中間にも障害物を置いて、「点」の障害物をなるべく建物のような「固まり」に近づけるルールになっています。

モデルを上書き保存してから、実行ボタンを押してみましょう。障害物が適当にばらまかれているでしょうか。もっともこの段階では、トリは障害物を無視して群を作るはずです（[図26.3](#)参照）。

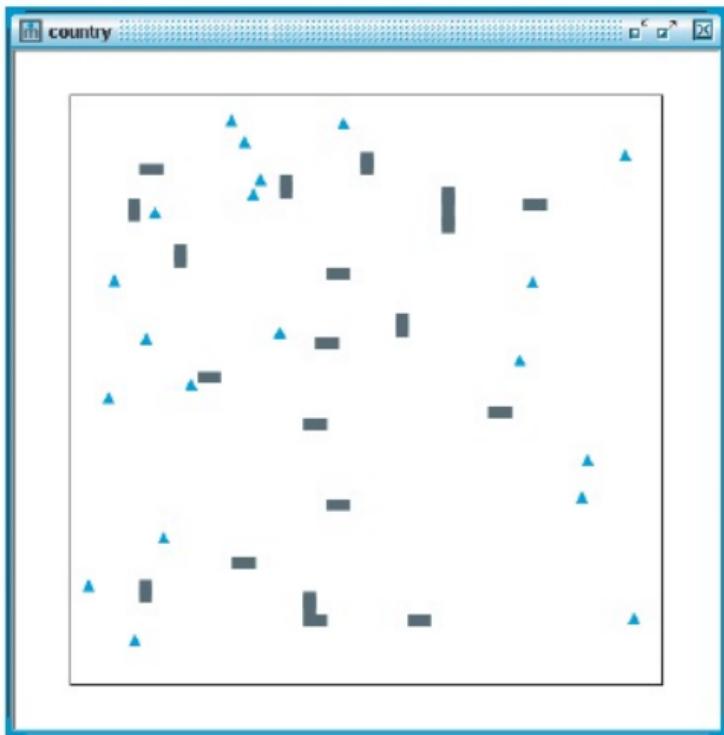


図26.3

26.4 ユーザ定義関数を作って、前方を調べさせる

それでは、前方にある障害物をトリが避けるルールを考えましょう。前方を調べる技法は、既に第11章で学びました。そこでは、同じルールを正面前方だけでなく左右前方を調べるために、繰り返して使っていましたが、ユーザ定義関数を使えば、もっとすっきりとルールを記述できます。

復習を兼ねて、前方を調べる技法を確認しましょう。birdエージェントの向いている正面前方(距離dist)の一一定範囲(range)に特定のエージェント種(type)がいるかどうかを調べる技法は、

```
Forward(dist)
MakeOneAgtsetAroundOwn(neighbor, range, type, False)
NoB = CountAgtset(neighbor)
Forward(-dist)
```

が基本です。NoBがゼロならエージェントはいなく、そうでなければエージェントがいます。

それでは前方を調べる作業を、ユーザ定義関数として定式化しましょう。ただし、障害物を置く作業SubBuildBarrier()と比べると、前方を調べる作業は1点異なります。調べた結果(NoBの値)を全体のルールの中で利用する必要があるのです。このような場合、Subではなく、Function(「ファンクション」と読みます)というタイプのユーザ定義関数を作ります。つまり、何かをさせる仕事をまとめるだけのときはSubを用い、仕事の結果を本文のルールで使用したいときはFunctionを用います。この関数の名前をWatchAheadとしましょう。

なおFunctionも、Subと同様に、ユーザ定義関数を使うルールエディタの最後に書き加えます。Function WatchAhead(){ }はトリ・エージェントのルールエディタの最後、つまりAgt_Step{ }のあとに続けます。具体的には、次のように定義します。

```
Function WatchAhead(dist As Double, range As Double, type As
Agttype) As Integer{
Dim neighbor As Agtset
Dim NoB As Integer
Forward(dist)
MakeOneAgtsetAroundOwn(neighbor, range, type, False)
NoB = CountAgtset(neighbor)
Forward(-dist)
Return(NoB)
}
```

Subと同様、Functionも作業を{ }で括ります。しかし、FunctionはSubと異なり、調べた結果(関数の値)を外に知らせる必要があります。そのために、WatchAhead() As Integerというふうに、関数値が「整数型である」と定義しておきます。具体的な値は、Return()により、Function WatchAhead(){ }中で行った操作の結果を{ }の外に知らせます。このユーザ定義関数では、一時的変数NoBがそれにあたります

が、NoBも整数型の変数でなくなりません。

そして、トリの行動ルールでは、

```
If WatchAhead(2, 0.7, Universe.country.barrier) > 0 Then
```

というふうに使います。WatchAhead()を実行すると、具体的にはNoBの値が戻っています。括弧の中は、2が前方距離、0.7が視野の範囲、そしてUniverse.country.barrierが調べる対象のエージェント種です。Function WatchAhead(dist As Double, range As Double, type As Agttype)の括弧内の変数ときちんと対応している必要があります。なお、調べる対象のエージェント種Universe.country.barrierに対応しているtype As Agttypeは、typeという変数がエージェント種別型であることを宣言する記述です。

この辺で、新しくboid(C)という名前を付けて保存しておきましょう。

26.5 障害物を避けるトリのルール

トリが障害物を避ける行動ルールを付け加えましょう。

前方に障害物があれば、それを避けるように曲がるが、また元の方向に戻る。

1. 正面前方を調べ、障害物があれば、左右どちらかの斜め(30度)前方を向く
2. その正面前方を調べ、障害物があれば反対側を向く
3. その正面前方を調べ、そこにも障害物があれば、左右どちらか90度向きを変える
4. 向いている方向に飛ぶ
5. 次のステップにはまた本来の飛ぶ方向に向いているようにする

ここで、1, 2, 3のルールは、向いている方向は異なるものの前方を調べるということについては同じです。ここに26.4で作ったユーザ定義関数Function WatchAhead()を使います。

一応、フローチャートを描いてみましょう。

[図26.4](#)を参考にして、次のようなルールを書き込みます。トリが飛ぶルールForward(My.speed)の直前で前方を調べます。なお、薄字のルールはboidモデルのルールとして記入済みです。

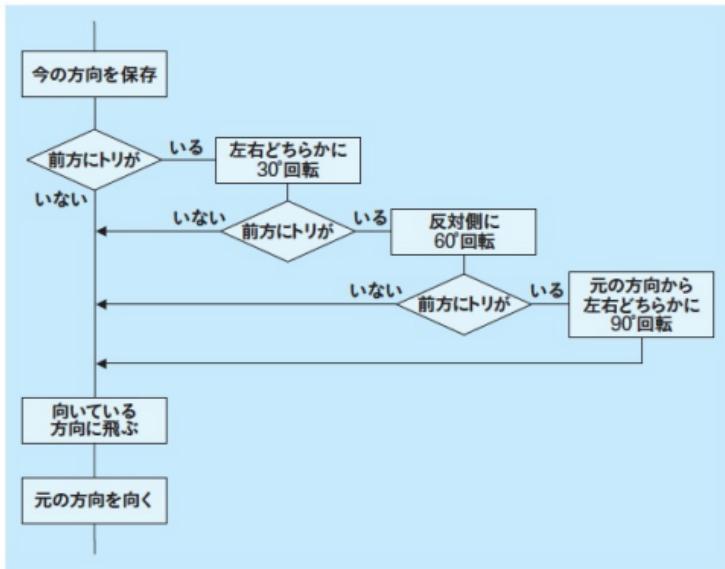


図26.4

```
Agt_Step{
Dim_deg As Double
Dim destin As Double
Dim neighbor As Agtset
Dim one As Agt
//群を作るルール
MakeOneAgtsetAroundOwn(neighbor, 2, Universe.country.bird,
False)
If CountAgtset(neighbor) > 0 Then //仲間がいれば
(中略)
End_if
//前方の障害物を避けるルール
destin = My.Direction //飛んでいく方向を保存
If WatchAhead(My.speed * 2, 0.7, Universe.country.barrier) > 0
Then
```

```

deg = (2 * Round(Rnd()) - 1) * 30 //30か-30になる
Turn(deg)
If WatchAhead(My.speed * 2, 0.7, Universe.country.barrier) > 0
Then
    Turn(-deg * 2)
    If WatchAhead(My.speed * 2, 0.7, Universe.country.barrier) >
0 Then
        Turn(deg + (2 * Round(Rnd()) - 1) * 90) //全方向に障害物があれ
        ば横を向く
    End if
End if
End if
Forward(My.speed) //向いている方向に飛ぶ
My.Direction = destin //方向を戻す
}

```

このルールにより、前方の障害物の有無に応じて、飛ぶ方向を一時的に変え、飛んだ後、再び、もともと飛んでいた方向に向き直ります。それでは、上書き保存して、実行してみましょう。



図26.5

障害物があちこちに配置されている空間を20羽のトリは障害物を避けながら飛ぶのと同時に、だんだんと大きな群を作る様子が観察できると思います。障害物を避ける過程で、群が分裂する場合もあるでしょう。

26.6 ユーザ定義関数を使い回す

上で作ったWatchAhead()というユーザ定義関数は、ボイド・モデルだけでなく、第11章で作ったターミナルで通勤客がすれ違うrush-hourモデルでも使えそうです。このように、ユーザ定義関数を複数のモデルで使いたいこともあるでしょう。ユーザ定義関数をartisocの「インクルード・ファイル」にしておけば、それが可能になります。

1. インクルード・ファイルを作る:

まずパソコンに備わっているテキストエディタを開いて、自分の作ったユーザ定義関数をそのままコピーし、たとえば、「WatchAhead」という名前のファイルとして保存して下さい。次に、このファイルの拡張子(この段階では.txtになっているはずです)を.incに変えて下さい。

2. インクルード・ファイルの中の関数をモデルで使う:

まずインクルード・ファイルをモデルのファイルと同じフォルダにしまって下さい。そしてエージェントのルールとして、このユーザ定義関数を使いたい場合は、エージェントのルールエディタの先頭(つまり、Agt_Init{ }の上の行)に、

```
include "WatchAhead.inc"
```

と書き込みます。あとは、組み込み関数と同じように使えます。

なお、インクルード・ファイルには、ユーザ定義関数をいくつも書き込んでおくことができます。関連するユーザ定義関数をまとめてひとつのインクルード・ファイルにしておくと便利です。

新しく学んだ事項

- ユーザ定義関数の作り方と使い方
- 仕事をさせるだけのときに使うSub
- 仕事の結果を知りたいときに使うFunction
- エージェント種別型変数(As Agttype)
- インクルード・ファイルの作り方と使い方

練習問題

同じユーザ定義関数を別な状況で使ってみましょう。

26.1

すぐ前を仲間のトリが飛んでいれば、速さを調節する(`My.speed = 0.2 + Rnd() * 0.3`)



`WatchAhead(My.speed, 0.5, Universe.country.bird)`

実行してみると、群の様子が少し変わるのがわかりますか？

26.2

障害物の大きさにバラエティをつける。基本の障害物を置き、その上隣または右隣に0.5だけ離して障害物を置く。さらに続けて最大8個になるまで確率的に障害物を置いていく



`BuildBarrier()`, フォ文とBreakの組み合わせ。

第27章

さまざまな技法を組み合わせる：ランダム・ネットワークを生成する

27.0 量から質へ転化します

- ランダム・ネットワーク生成のモデルを作ります
- エージェント集合, ID, 配列などの利用に慣れましょう
- 円形配置, 線引き, 色づけ, 最大値探しなど応用技法が盛りだくさんです
- さまざまな技法を組み合わせるとモデルが飛躍します

27.1 ランダム・ネットワークの基礎

エージェント集合, IDによる個体識別と配列を利用した技法に慣れるために, 簡単ですが大変に興味深いモデルを作りましょう。それはランダム・ネットワークが生成されていくモデルです。

ネットワークは, ノード(頂点)とノードとノードとを結ぶリンク(枝)から成り立っています。ノードをエージェントと見なすと, リンクはエージェント間の相互作用の現れです。したがって, ネットワークが生成されていく過程は, マルチエージェント・シミュレーションで表すことが可能です。近年, 啓蒙書が何冊も刊行されたこともあって, ネットワークに注目が集まっています。スケールフリー・ネットワークやスマールワールド・ネットワークといった言葉を聞いたことのある人も多いと思います。

今まで学んだartisocの技法で, このような複雑なネットワークの生成をモデル化することが可能ですが, まずは, 最も基本的なランダム・ネットワーク生成のモデルを作ることにしましょう。

多数のエージェントが互いにランダムに結びつけられていく。

モデル化の基本はこれだけです。それでは準備作業から始めましょう。

1. Universeの下に空間fieldをデフォルトのまま追加し、その下にエージェントnode(エージェント数は0)を追加する
2. Universeにはnodeの数を決めるnumnodesと各ステップで実行するエージェントを決めるselectをどちらも整数型で追加する
3. nodeにはリンクの相手をリストアップするエージェント集合型変数linkを追加する
4. numnodesの値を決めるコントロールパネルを設定する。スライドバーで20から100まで、10刻みで設定できるようにする
5. マップ出力を設定する。linkを線引対象にする
6. モデルをrndnetという名前を付けて保存する

これで準備完了です。

27.2 ランダムにリンクをはっていく

ルールとしては、まずUniv_Init{ }でエージェントを生成し、Univ_Step_Begin{ }で、各ステップ、リンク相手を捜すエージェントを選別します。言い換えると、1ステップにリンクを1つだけはることにします。

```
Univ_Init{
Dim i As Integer
For i = 0 To Universe.numnodes - 1
    CreateAgt(Universe.field.node)
Next i
}
Univ_Step_Begin{
    Universe.select = Int(Rnd() * Universe.numnodes)
}
```

各エージェントのルールは次のようになります。まず、ノードの初期配置です。円形に等間隔で配置する技巧的なルールになっています。

```
Agt_Init{
MoveToCenter()
```

```
My.Direction = 360 * My.ID / Universe.numnodes
Forward(20)
ClearAgtset(My.link) //念のため初期化
}
```

MoveToCenter(), My.Direction =, Forward()の組み合わせは、汎用性のある円形配置のルールです。

次に、各ステップ、自分がUniv_Step_Begin{}で選抜されているときだけ、ルールを実行します。

```
Agt_Step{
Dim one As Agt
Dim set As Agtset
If My.ID == Universe.select Then //自分が選ばれたら
//自分以外のノードからランダムに1つ選んでリンクをはる
  MakeAgtsetSpace(set, Universe.field)
  RemoveAgt(set, My) //自分を除く
  one = GetAgt(set, Int(Rnd() * (Universe.numnodes - 1)))
  AddAgt(My.link, one)
  AddAgt(one.link, My) //相手側にも自分を入れる
End if
}
```

これで完了です。RemoveAgt()は初出ですが、AddAgt()の逆に、エージェントをエージェント集合から取り除く関数です（第24章の練習問題のヒントで登場しました）。上書き保存してから、実行してみましょう。何も起こらないときには、いったん停止ボタンを押し、コントロールパネルを操作してから再び実行して下さい。たとえば、ノードの数を40にすると、10ステップ目、30ステップ目、60ステップ目は[図27.1](#)のようになります。

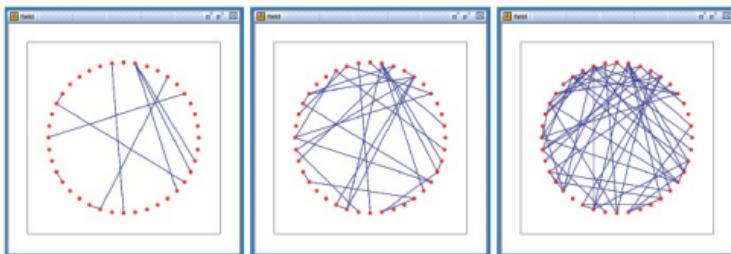


図27.1

厳密に言うと、上のルールでは、既にはられたリンクが再びはられてしまう可能性があります。そのような場合を除外するルールを追加することは可能ですが、リンクのはり方の骨子を示すために、敢えて単純化してあります。つまりこのモデルは、ランダム・グラフの定義を厳密に再現しているというよりは、リンクをランダムにはっていくことによってネットワークがダイナミックにできていく過程をモデル化したものです。

27.3 ランダム・ネットワークのクラスターに注目

ランダム・ネットワークの生成では、リンクがだんだん増えていくと急に大きなクラスターが形成されるという現象が特に注目されています。クラスターとは、リンクによって互いに結びつけられたノードの集合のことです。

そこで、どのようなクラスターが形成されているのかを

1. 同一クラスターに属すノードは同一色でマップ上に表す
2. クラスターの数を時系列グラフに示す
3. 最大クラスターに含まれるノードの数を時系列グラフに示す

という3つの角度から把握することにしましょう。1については、エージェントのルールで、2と3については、Universeのルールで処理します。以下でも、技巧的なルールをいくつか学びます。

27.4 クラスターを区別して表す

まず、nodeの下に、自分が属すクラスター(識別番号と色)を表す整数型変数clusIDとcolorを追加します。マップ出力設定も編集作業をして、要素の色変数指定(もちろんcolorを選択)をしてください。

Agt_Init{ }のセクションの最後に初期設定を追加します。最初はリンクがないので、各ノードがそれぞれ別個のクラスターを構成しています。そこでノードのIDをそのまま使ってクラスターを識別しておきます。同様に、各ノード=各クラスターに色をランダムに割り振ります。

```
ClearAgtset(My.link) //念のため初期化  
My.clusID = My.ID  
My.color = Int(Rnd() * Color_White) //色のスマートな指定  
}
```

色のランダムな割り振り方については、コラム[「色の秘密」](#)を参照して下さい。

Agt_Step{ }には、リンクをはった後で、クラスターを見直します。もしリンクをはった相手のクラスターが自分のクラスターと異なれば、その相手のクラスターを自分のクラスターに併呑します。リンクの相手だけでなく、その相手が既にリンクをはっているノード全てを自分のクラスターに入れる点に注意して下さい。ルールは、一時的変数の宣言文以外は、全体のイフ文の中の最後尾に挿入します。

```
Dim num As Integer //冒頭に追加  
  
//新しくリンクをはる相手のクラスターが自分のクラスターと異なれば  
If My.clusID <> one.clusID Then  
    num = one.clusID  
    //その所属クラスターのノード全てを自分のクラスターに入れる  
    For each one in set  
        If one.clusID == num Then  
            one.clusID = My.clusID  
            one.color = My.color //色も合わせる  
        End if  
    Next one  
End if  
  
End if //既に書き込まれているIf My.ID == Universe.select Thenの締め括り  
}
```

それでは、rndnet(B)という名前を付けて保存し、実行してみましょう。同一クラスターが同一色になっているはずです。

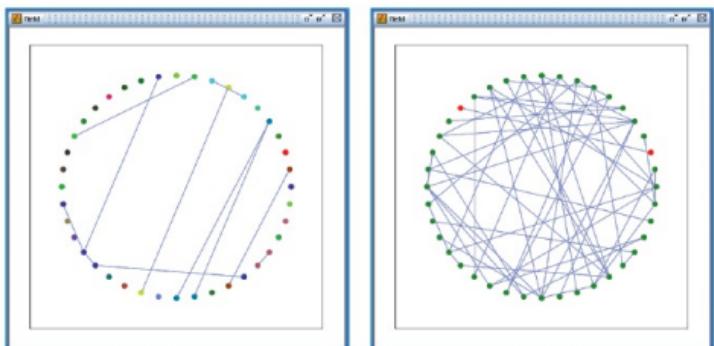


図27.2

27.5 全体像の特徴を把握する

最後に、クラスターの状況を把握するための作業をします。

1. クラスター数と最大クラスターに含まれるノード数を集計・出力するため、Universeにnumclustersとmaxclusterという変数をどちらも整数型で追加して下さい
2. 時系列グラフを出力設定する。numclustersとmaxclusterをそのまま出力するようにして下さい。横軸、縦軸の目盛りの設定も適当に直しましょう。

とりあえず、Univ_Init{ }に、初期値を入れておくルールを追加します。これは時系列グラフの見栄えをよくする(正確な値を出力させる)ためです。

```
Univ_Init{
Dim i As Integer
For i = 0 To Universe.numnodes - 1
  CreateAgt(Universe.field.node)
Next i
Universe.numclusters = Universe.numnodes
Universe.maxcluster = 1
}
```

クラスターがどうなっているかを把握するルールは下記のように、長くて、やや複雑です。最後に、終了条件のルールも書き込んであります。

```
Univ_Step_End{
Dim i As Integer
Dim clus(100) As Integer //配列のある一時的変数を宣言
Dim one As Agt
Dim set As Agtset
//初期化
For i = 0 To 99
    clus(i) = 0
Next i
Universe.maxcluster = 0
Universe.numclusters = 0
MakeAgtsetSpace(set, Universe.field)
//各ノードが属しているクラスターに加算
For each one in set
    clus(one.clusID) = clus(one.clusID) + 1
Next one
//クラスター数と最大クラスターのノード数を調べる
For i = 0 To Universe.numnodes - 1
    If clus(i) <> 0 Then //ノードのあるクラスターについてのみ
        Universe.numclusters = Universe.numclusters + 1
        If Universe.maxcluster < clus(i) Then //常に多い方を保存
            Universe.maxcluster = clus(i)
        End if
    End if
Next i
//クラスターが1つになつたらシミュレーションを終了させる
If Universe.numclusters <= 1 Then
    ExitSimulationMsgLn("All nodes in a cluster after" &
GetCountStep() & "steps")
End if
}
```

文法的に新しい技法は

```
Dim clus(100) As Integer
```

だけです。これは、配列を持っている一時的変数を宣言する方法です。clus(100)で要素番号0～99(0～100ではなく)の要素数100の配列が宣言されています。もし、100×20の2次元配列tempを実数型の一時的変数として宣言したい場合は、

```
Dim temp(100, 20) As Double
```

のように表記します。

なお, clus(100)の配列数が100になっているのには理由があります。27.1でノードの最大数がコントロールパネルで100に設定されているために、クラスターの最大数も100になるからです。ツリーの変数だろうと一時的変数だろうと、配列数の上限を越えるようなルールが実行されるとエラーになります。

それでは、rndnet(C)という名前を付けて保存して、実行して下さい。クラスターの様子が変化していくのが見て取れるはずです([図27.3](#)参照)。

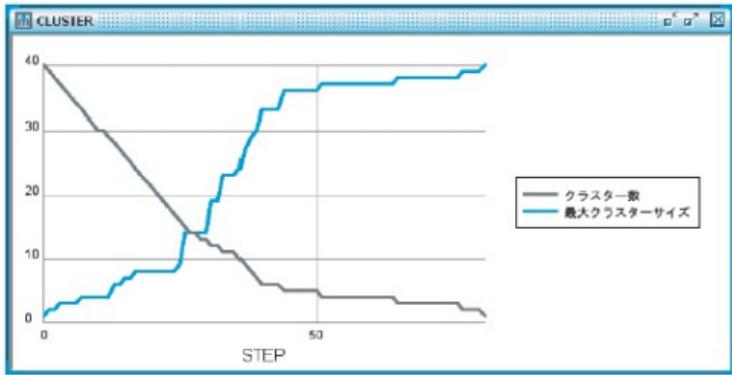


図27.3

新しく学んだ事項

- エージェントを円形等間隔に配置する技法
- 配列のある一時的変数の宣言
- RemoveAgt()

練習問題

この章ではいろいろと技巧的な手法を学んだので、復習を兼ねて、IDと配列の組み合わせについての考え方慣れましょう。

27.1

リンクをはる相手を選ぶルールは、エージェント集合を操作してGetAgt()によって選び出す代わりに、エージェントIDを使って指定することもできる。rndnet(C)モデルのAgt_Step{ }のセクションを、IDによって相手を選び出すルールに書き換えてみよう。



練習問題25.2の応用です。自分自身のIDが選ばれないようにしましょう。

27.2

ランダム・ネットワークのモデルで、クラスターの集計ルールの部分の下記の表記が技巧的でわかりにくかったかも知れない。

```
For each one in set
  clus(one.clusID) = clus(one.clusID) + 1
Next one
```

クラスターの様子がなぜこれで把握できるのか、エージェントのルールも含めて、全体のロジックを考えよう。

第28章

多様な空間構造を利用する

28.0 格子や2次元にこだわることはありません

- artisocの空間設定にはもっと自由度があります
- 格子(四角形)ではなく六角形も使えます
- 階層化することも可能です

28.1 空間構造の多様性

これまでのartisocのモデルは空間が二次元で、エージェントの位置はX座標とY座標を用いてルール化してきました。それでも、空間の構造をいろいろなタイプにすることができました。第1に、空間がループしているか、していないかによって、空間の性質は大きく異なりました。第2に、空間が連続的で、エージェントが自由に動ける場合と、空間が格子のかたちをしていて、エージェントは格子上をとびとびにしか動けない場合とで、やはり空間の性質は大きく異なりました。

以上のようなタイプの違いを組み合わせるだけで、さまざまなモデルを作ることができます。実際、本書でもさまざまなタイプの空間を考えて、モデルを作ってきました。

しかしartisocには、さらに空間構造を多様にできるオプションがあります。そのひとつが、格子型の空間を正方形のセルではなく、六角形のセルにする方法です。これを「六角モデル」といいます。そして、もうひとつが、2次元空間(格子モデルで連続的だろうと離散的だろうと、六角モデルだろうと)を重層的にするレイヤというオプションです。

この章では、六角モデルの空間構造を利用したモデル例と、レイヤを利用したモデル例を紹介します。

28.2 六角モデルとは

領土を取り合ったりするボードゲームでは、世界は小さな正六角形がたくさん敷き詰められていることが多いります。このように、空間を多数の六角形のセルに切り分けた構造をartisocでは「六角モデル」と呼び、このような空間構造を選ぶことができます。空間をUniverseの下に追加するとき、今まで使ってきたデフォルト値の「格子モデル」ではなく、「六角モデル」を選択するだけです。

六角モデルの空間構造を用いてセル型モデルを作成すると、セルは蜂の巣状に並び、辺どうしが接して、周囲は必ず6セルです(閑話休題「[見方を変えると周りも変わる](#)」参照)。また、「周囲」も六角モデルでは円に近い形になり、私たちが直感的に周囲と見なす範囲に近くなり、「自然」に見えます。

このような空間構造は、エージェントが自由に空間を動き回るタイプのモデルではなく、エージェントがマス目に沿って動いたり、地理的単位をエージェントとするマルチエージェント・シミュレーションのモデルに適していると言えるでしょう。特に複数のエージェントがさまざまな配置でまとまり、地域社会とか国家のような領域を構成し、複雑な空間的近隣関係を生み出すような現象のモデル化に六角モデルの空間構造は便利です。

28.3 帝国モデルの概要

ここでは、六角モデルに設定された空間を使ったモデルの例として、次に述べる帝国モデルを作成します。六角モデルの特徴を示すのが目的なので、モデル自体はきわめて簡単にしています。

初期状態として、100の国家がそれぞれ1領土単位を持っている。毎ステップ、領土を拡大しようとする国が一国選択される(領土がより大きな国のはうが選択されやすい)。選択された国は、必ず自分に隣接する国の領土の1単位を獲得する。

それではモデル作りに取りかかりましょう。

1. Universeの下に、 10×10 のループしない空間landを作ります。今回は、「空間プロパティ」の「空間

種別」で、「六角モデル」を選択します。

2. 空間landの下に、各領土単位unitエージェントを追加し、「エージェント数」を0(デフォルト値)とします。Universeの初期設定で、100の領土単位を生成します。
3. unitエージェントに、所属する国家を表す色を格納するための整数型変数stateと所属する国家を識別するための整数型変数SIDを追加します。初期状態では、1つの領土単位が1国家を構成するとしています。
4. Universeの下にどの国家が攻撃国に選択されたかを記録する整数型変数agrsを作成します。
5. 出力設定で、マップ出力を行ないます。領土単位unitを「マップ要素リスト」で出力対象とし、「エージェント表示色」→「変数指定」で、変数stateを選択します。また、マップのXとYの最大値とともに9として下さい。
6. このモデルをempireという名前を付けて保存して下さい。

28.4 帝国モデルのルール記述

1. Univ_Init{ }で初期設定をしましょう。領土単位を空間landにランダムに配置します。上に述べたように、初期状態では、1つの領土単位が1国家を構成するので、領土単位毎に個別の国家SIDとstateを設定し、各領土単位をランダムに配置します。

```
Univ_Init{
    Dim i As Integer
    Dim one As Agt
    Dim set As Agtset
    For i = 0 to 99
        one = CreateAgt(Universe.land.unit)
        one.state = Rnd() * Color_White //色をランダムに指定
        one.SID = one.ID
    Next i
    MakeAgtset(set, Universe.land.unit)
```

```
RandomPutAgtsetCell(set, False)
}
```

それでは上書き保存して、実行して下さい。うまく国家が初期配置されたでしょうか。うまくいったなら、停止ボタンを押して、続けて作業します。

- 各ステップで攻撃を行なう国家を決定します。ここでは、攻撃国をランダムに選択します。ルールは Univ_Step_Begin{ }に記述します。

```
Univ_Step_Begin{
    Universe.agrs = Rnd() * 100
}
```

- いよいよ、国家の行動ルールです。ここでのポイントは、エージェントが国家ではなく、領土単位になっている点です。攻撃国は、上のルールで領土単位が選ばれますから、その領土単位を領有している国家になります。そして、その国が自国の隣接国から1つの領土単位を獲得するルールを記述します。次のルールをAgt_Step{ }に記述して下さい。

```
Agt_Step{
    Dim flg As Boolean      //各ステップでの併合完了を知らせる
    Dim one As Agt
    Dim cand As Agt
    Dim set As Agtset
    Dim neighbor As Agtset
    If My.ID == Universe.agrs then      //攻撃国である場合
        flg = False      //併合完了をチェックするための変数の初期化
        MakeAgtset (set, Universe.land.unit) //領土単位を全部集める
        For each one in set //全ての領土単位について自分の領土かチェック
            If one.SID == My.SID Then      //自分の領土であった場合
```

```

    MakeAllAgtsetAroundOwnCell(neighbor, 1, False) //その周
    囲について

    For each cand in neighbor //1つずつチェック
        If cand.SID <> My.SID Then //他国の領土なら併合
            cand.SID = My.SID
            cand.state = My.state
            flg = True //併合に成功
            Break //1つ併合したら、それで完了

        End if

        Next cand
        //Breakでここに出る

    End if

    If flg == True Then //併合完了ならおしまい
        Break
    End if

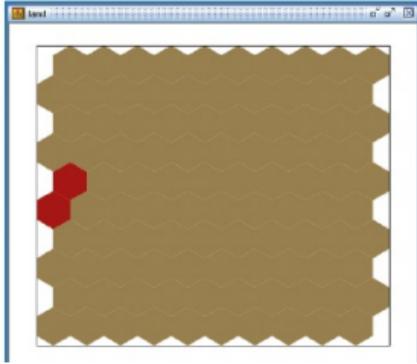
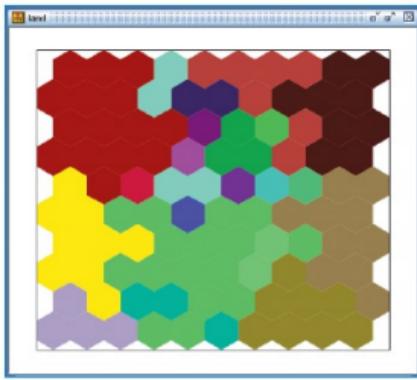
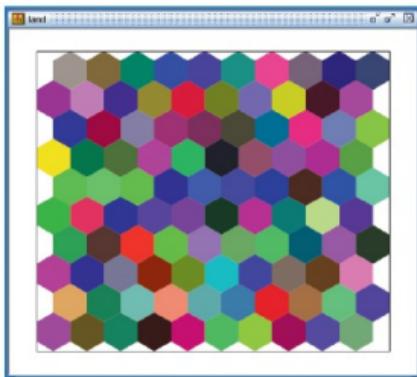
    Next one
    //Breakでここに出る

End if
}

```

上のルールでは、攻撃国であった場合、自分の領土に隣接する他国の領土が存在するかを順次チェックしていく、最初にみつかった他国の領土を自国に編入するというルールになっています。現実の戦争では、どの国を攻撃するかは軍事力や友好関係などのさまざまな要因で決定され、また戦争の勝敗も攻撃国と攻撃された国どちらが勝利するかは、簡単には決定されません。しかし、ここでは、これらの複雑な現実はモデル化していません。単純に、必ず攻撃国が勝利するとし、攻撃の対象となった国家の領土単位を1つだけ獲得するとしています。

それではempire(B)という名前で保存してから、実行して下さい。



これで、帝国モデルが完成しました。攻撃国決定のルールで、各領土単位のID(SIDではない)をもとにして、一様に攻撃国を決定しているので、領土が大きい国家ほど、より攻撃しやすくなっています。また、攻撃すると必ず領土を1単位拡大するので、領土を拡大した国家ほど大きくなりやすくなっています。したがって、このモデルでは、やがて空間全体を支配する1つの帝国が誕生することになります。

このモデルのように、六角モデルの空間構造の上で地理的単位をエージェント(動かないセル)になると、複数の単位が1つの上位の単位(ここでは国家)にまとまっていく様子の「見栄え」が自然になります。四角形のセル(「格子モデル」設定)では、角だけが接する斜め方向の領土を取得する場合も生じてしまい、国土が奇妙な広がり方をします。

28.5 レイヤとは

artisocの空間にはレイヤ(=階層)というもうひとつの軸が備わっています。この軸を利用すると、二次元空間(格子モデルでも六角モデルでも)が何層にも折り重なって構成されているような、レイヤ(階層)構造をもった空間が表現できるのです。

これまでエージェントの空間上の位置は、X値とY値のみで指定されているかのようでしたが、実際にはエージェントの位置は常に、X(実数型)、Y(実数型)とLayer(整数型)の3つの値によって指定されているのです。ツリー上の空間にエージェント種を作成したとき、必ずLayerという変数も自動的に作成されますが、これまで作ってきたモデルの空間では、レイヤの数はデフォルト値の1のままで、全てのエージェントのレイヤの値は常に0でした。

レイヤは、次のような特徴を持っています。

- 異なる階層にいる他のエージェントは、XY座標が近くても認識することはできない(一層のみの空間では、こういう関係を表現するのは難しい)
- エージェントは階層間を移動することができる(複数の空間では、エージェントがその間を移動する

のは不可能である)

この特徴をうまく用いれば、単なるひとつの二次元空間でもなく、別々に存在する複数の二次元空間でもない複雑な空間を表現することができます。たとえば、複数の文化圏に属する人々の相互作用はどうでしょうか。人々は基本的にはそれぞれの文化圏内で生活しています。しかし、他の文化圏に移動する人々がいないわけではないし、文化圏間で相互作用がないわけではありません。このような複雑な空間表現を行なうのに、レイヤは非常に便利です。

レイヤを設定して、階層構造を持つ空間をつくるのは簡単です。空間を最初に作るときに、X軸やY軸の大きさを設定するときと同様に、レイヤ数を入力してやれば、その数だけの階層が形成されます。空間を作ったあとでも、プロパティ・ウィンドウでレイヤ数を変更すれば、好きなだけの階層構造を作ることができます。

28.6 人事モデルの概要

これから、レイヤの特徴を活かした「人事モデル」を作成してみましょう。

レイヤを利用して空間を階層化することのメリットは、同じ性質の空間が何段階かに分かれている状況をモデル化できる点です。ここでは、平社員、管理職、役員の3階層からなる会社をモデル化します。レイヤを使うことが主眼ですから、モデルは単純なものにしましょう。

二つの分野(X, Y)で特徴づけられる会社がある。会社は平社員、管理職、役員の3層に分かれている。10人が会社を設立し(全員が役員)、その後、毎ステップ(毎年と考えます)新人(平社員)を採用する。成績に応じて昇進するが、定年が来れば退職する。

1. Universeの下に、空間company(レイヤを3に設定、他はデフォルトのまま)を追加します。レイヤ0の層は平社員の層、レイヤ1の層は管理職の層、レイヤ2の層は役員の層を表すものとします。X軸とY軸は、会社の仕事の何らかの分野を表すものとします。近しい分野ほど、XY座標も近いということです。

2. companyにエージェントmember(エージェント数は0)を追加し、それに年齢を数と色で表す整数型変数ageとcolor、そして成績を表す整数型変数pointを追加します。エージェントはUniverseの初期設定で生成します。
3. マップ出力設定をします。マップ出力は1つにつき空間1層しか表せません。複数のレイヤのマップを出力したい場合は、「出力設定」の「出力項目リスト」でレイヤ毎にマップ出力を追加します。具体的には「マップ出力設定」のウインドウで、「空間名」の下にある「表示階層」にレイヤの値を書き込むことで、そのレイヤを表示させることができます(デフォルト値は0になっています)。それでは、全レイヤを出力することにしましょう。マップ名やマップタイトルにそれぞれ異なる名前を付けて(たとえばworkers, managers, executives)レイヤを区別して下さい。全ての階層で、マップ要素としてmemberエージェントを選び、変数colorによる色指定にしておいて下さい。それ以外はデフォルトでかまいません。
4. このモデルをpersonnelという名前で保存して下さい。
ためしに一度実行してみて下さい。3つの二次元空間マップが重なり合って表示されたことと思います。一度シミュレーションを停止して、**メニュー** > **ウインドウ** > **並べて表示** を選択してみて下さい(さらに今回は邪魔なのでコンソール画面を消しておくと、なおよいです)。3つの階層がきれいに並べて表示されたはずです([図28.2](#))。

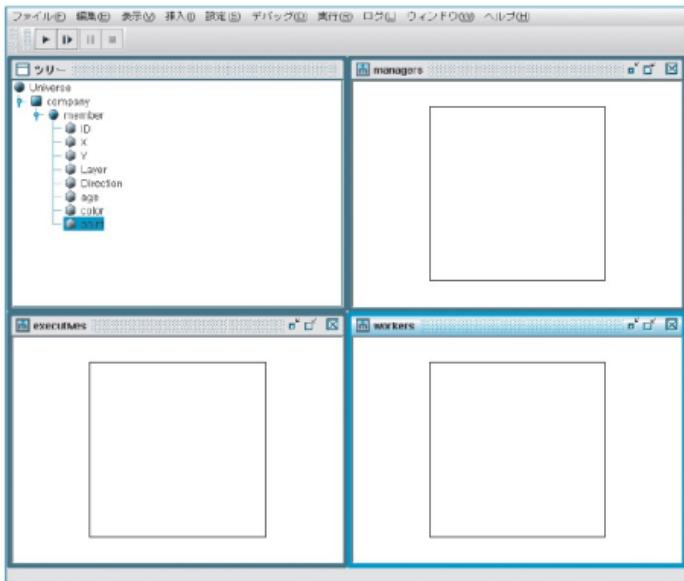


図28.2

28.7 人事モデルのルール記述

- 会社を立ち上げる最初の10人のメンバー（全員、役員です）は、大学を卒業したての同級生（年齢はみんな23歳）で、自分の得意な仕事の分野はバラバラだったとしましょう。エージェントの色は若いときは青く、加令とともに赤っぽくなるように設定しましょう。エージェントの生成と初期設定は以下のとおりです。Univ_Init{ }とAgt_Init{ }とに分かれています。

```
Univ_Init{
    Dim i As Integer
    Dim one As Agt
    For i = 0 to 9
```

```

one = CreateAgt(Universe.company.member)
one.Layer = 2 //全員、役員
//他の変数値はAgt_Init{ }で設定

Next i
}

Agt_Init{
My.X = Rnd() * 50
My.Y = Rnd() * 50
My.age = 23
My.Layer = 0 //創業者たちはUniv_Init{ }で上書き更新
My.color = RGB(0, 0, 255)
}

```

Univ_Init{ }のルールでCreateAgt()が実行されると直ちにAgt_Init{ }のルールが実行され、再びUniv_Init{ }に戻るので、上のようなルール記述でうまくいきます([31.3](#)を参照)。上書き保存して実行してみましょう。役員の層に10人のメンバーが表示されているはずです。彼らの色が青く、若々しいことと思います。

2. エージェントの行動ルールの大枠を書きます。普通にエージェントのルールを書くと、どの階層にいるエージェントにあっても共通になってしまいます。そのため、ある階層にいるエージェントにだけ実行してほしいルールは、あらかじめ場合分け(条件分岐)を用いて、そのことを指定しておかなければなりません。たとえば、Agt_Step{ }のルールにおいては、以下のように書くことが必要です。

```

Agt_Step{
If My.Layer == 0 Then
    //平の従業員だけに適用されるルール
Elseif My.Layer == 1 Then
    //管理職だけに適用されるルール
Elseif My.Layer == 2 Then //Elseだけでもよいが、念のため

```

```
//役員だけに適用されるルール  
End if  
//すべての社員に適用されるルール  
}
```

3. 全ての社員に適用されるルールとして、毎年、みんなが年を取るというルールがあります。そして、それに伴い、色も変化させましょう。さらに、定年（たとえば63歳）が来れば、退職して会社を去ります。

```
My.age = My.age + 1  
My.color = RGB((My.age - 23) * 6, 0, 255 - (My.age - 23) * 6)  
If My.age >= 63 Then  
    KillAgt(My) //そのステップの最後に（一斉）退職  
End if
```

上のルールで、KillAgt(My)という表記が出てきました。定年退職は「老兵は死なず、ただ消えゆくのみ」という表現がぴったりですが、ぶっそうなKillが使われています。KillAgt()はDelAgt()と同様にエージェントを抹消する関数ですが、エージェントの消し方に若干差があります。詳しくはコラム【[エージェントを消す三つの関数](#)】を参照して下さい。

それではpersonnel(B)という名前を付けて保存してから、試しに実行してみてください。みんなだんだん年をとって（赤っぽくなつて）、退職していくのでしょうか。

4. 各ステップに一人の新人社員が2名リクルートされることとしましょう。Univ_Step_Begin{ }のルールに以下のように書きます。

```
Univ_Step_Begin{  
CreateAgt(Universe.company.member)  
CreateAgt(Universe.company.member)
```

```
}
```

新人は、CreateAgt()の実行時には直ちにAgt_Init{ }が実行されるので(Univ_Init{ })は実行されないので、年齢は常に23歳、得意分野はランダムに与えられ、平社員のレイヤに登場します。

5. 次に出世のルールを導入しましょう。成績に応じて昇進するものとします。ここで、各memberの成績は、役員がそのmemberと一緒に作業をする近しい同僚の数(動員力? 人望?)でもって評価するものとしましょう。そして、成績はだんだん累積されていくって、ある一定の値(たとえば30)を超えると中間管理職へ、そして役員へと出世していくこととしましょう。

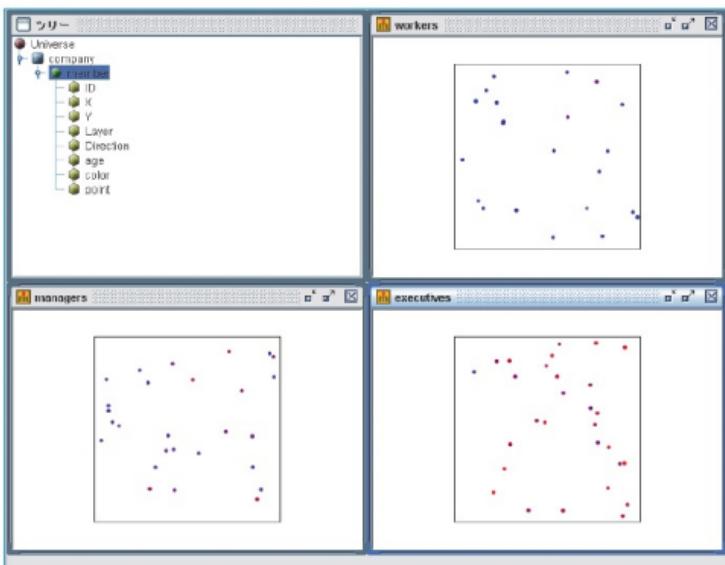
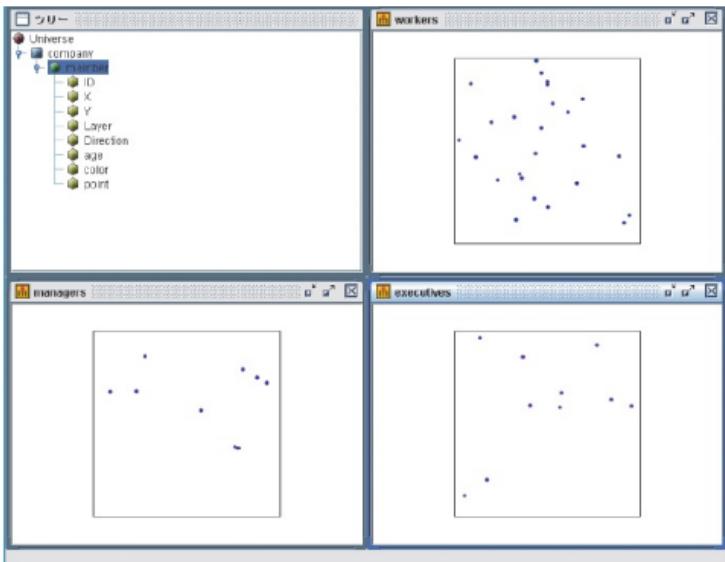
```
Dim colleagues As Agtset //Agt_Step{ }の冒頭に追加
```

```
//昇進のルール(平と管理職の両方に適用)
```

```
MakeAllAgtsetAroundOwn(colleagues, 10, False)  
My.point = My.point + CountAgtset(colleagues)  
If My.point >= 30 Then  
    MoveLayerSpace(1)  
    My.point = 0  
End if
```

このルールは、平の従業員にも管理職にも適用されるので、両方に書いておいて下さい。ここで、新しいルール表現MoveLayerSpace()が登場しました。これは、移動させたいレイヤの数(上のルールではプラス!)だけレイヤを移動させるルールです。負の値を設定すればレイヤの値は下がります。

それでは、personnel(C)という名前で保存して実行してみましょう。平社員が中間管理職に出世し、役員に出世する様子がわかると思います。よくみると、平社員のまま、年を重ねていく従業員もいますね([図28.3](#)参照)。



新しく学んだ事項

- 空間構造を六角モデルにする
- 空間を階層化する
- マップに空間を階層毎に出力する
- エージェントの変数Layerの使い方
- MoveLayerSpace()
- KillAgt()

練習問題

六角モデルや階層(Layerの利用)に慣れましょう。

28.1

六角モデルの構造を持った空間では、セル上におけるエージェントの動き方が格子モデルとは異なる。周囲が8近傍ではなく6近傍なので当然だ。どんな動きをするだろうか。第9章で作ったrealcellモデルを参考にして、六角モデルで同様のモデルを作ってみよう。

六角モデルの構造を持った空間(大きさ 6×5)の上を1つのエージェントが1セルずつ(右横、斜め右上方向、斜め右下方向)に移動する。



ForwardDirectionCell()を使う点は、格子モデルのときと同じなのですが。

personnel(B)モデルでは、管理職の数や役員の数は一定しておらず、たくさんになったり少なくなったりと不安定である。それぞれの役職に定員があるものとしよう。

管理職と役員には定員がある(各々20人と10人)。定員に空きがあるときのみ出世できる。



取り敢えず昇進してみて、空きがないなら……。

Column

エージェントを消す三つの関数

第16章でエージェントを抹消する(殺す)ための技法としてDelAgt()関数を学びました。しかし、似たような機能を持つ関数として、KillAgt(), TerminateAgt()という別の二つの関数があります。いずれも、1ステップのうちに、()内で指定されたエージェントをまるまる消し去ってしまうという点で同じ機能を持っていますが、抹消のタイミングや方法において若干の違いがあります。以下に各関数の働き方の違いを、簡単にまとめておきましょう。

KillAgt()関数

この関数は、実行された時点で、抹消されるエージェントに特別なフラグ(Killフラグ)を立てます。

ステップ内にフラグを立てられたエージェントは、ステップ終了時(Univ_Step_End{})のルール終了後)にまとめて消去されます。つまり、この関数では、エージェントは関数の実行時点では消去されず、消去はステップ終了時まで先延ばしされることになります。

Killフラグを立てられたエージェントは、ステップ終了するまで人工社会の中に「存在」しています。この間に実行順序が回ってくれればルールは実行され、また他のエージェントが作用を及ぼすこともでき

ます。このような死者との相互作用を避けたい場合、SpecifyKillAgt()という関数を用いてエージェントの生死を判別します。Killフラグが立っているエージェントの場合、この関数の値はTrueになります。

DelAgt()関数

これに対して、DelAgt()関数は、関数実行時点で即座に指定されたエージェントの存在を抹消します。第16章で見たように、この関数で消去されたエージェントは、CountAgtset()関数等によってもカウントされず、またルールが実行されることも、他のエージェントから作用を受けることもありません。

それでは、DelAgt()関数によって、エージェントの即時・完全な抹消が実現されるのかというと、必ずしもそうではありません。たとえばエージェントが持っている(つまりツリーに作った)エージェント集合型変数の中に、DelAgt()関数で抹消された他のエージェントが要素として入っていた場合、集合における後者の「痕跡」(厳密にはそのエージェントのID)が抹消後も消されず残っています。そのため、Universeやエージェントに追加した種々の変数(エージェント型・エージェント集合型)の操作を通じて、死者を(生きているものとして)参照してしまうという可能性が、DelAgt()関数の場合にも残されていることになります。

TerminateAgt()関数

このようなモデルに残されるうる様な「痕跡」も含めて、指定したエージェントの存在を即時に、かつ完全に消去するのが、TerminateAgt()関数になります。

消し方を使い分けよう

つまり、KillAgt() < DelAgt() < TerminateAgt() の順に、エージェント抹消の完全さが増していくことになります。同時に、このことは、関数実行に伴う処理の複雑さ、つまりはartisocによるシミュレーション実行への負荷が、同じ順序で増していくことも意味しています。特に、TerminateAgt()関数は、抹消の対象となるエージェントだけではなく、他の全てのエージェントが持っている変数までもチェックするため、使用するモデルの内容によっては、目に見えて実行速度が遅くなる場合も生じます。

す。したがって、モデルの中でエージェントを抹消する場合には、エージェント間の相互作用の内容といった点に加え、シミュレーションの実行速度等も考慮して、三つの中から適切な関数を選ぶ必要があります。

(阪)

現代人(?)はゼロから数える

20世紀末(といつても数年前ですが), 21世紀は2000年からなのか2001年からなのかが話題になりました。正解は2001年です。ところで西暦の原点とも言えるイエス・キリストの誕生年は後年の研究で西暦紀元1年ではなく、紀元前4年らしいと言われていますが、仮にそうだとすると、2006年から見て何年前でしょうか。2006-(-4)だから2010年前……間違いです。正解は2009年前です。この種の問題は、暦を作った人たちが出発点をゼロではなく1にしたことから起きました。ゼロという数を知らなかった(ないし拒絶した)からです。

私たちは日常生活でゼロから数えるのか1から数えるのか混乱しています。たとえば、日本語なら2時20分と言うところをドイツ語では3時間目の20分と言います。昔、日本では赤ちゃんが生まれるとすぐ1歳になり、正月を迎えるたびに1歳ずつ年をとりました。今では、生まれてから最初の誕生日の前々日までは、ゼロ歳児と言います(満年令は、誕生日の前日に1つ増えます。これにより2月29日生まれの人も1年に1歳ずつ年をとることができます)。しばらく前までは「数えで8歳、満7歳」という風に混乱を避ける言い方をしていました。ちなみに厄年は「数え」で数えます。

そこで整理が必要になります。順番を数えるための数を序数と言います。最初(1番目)は何もない状態(仮定)から出発するので、ゼロ(0)から始まり、1, 2, 3……と続きます。ですから、四番目に対応する数は3です。4ではありません。他方、いくつあるのかを数えるための数を基数と言います。何もなければ(誰もいなければ)、ゼロ個(人)です。続いて、1個(人), 2個(人), 3個(人), ……と続きます。序数と基数とでは考え方方が異なりますが、同じ数字を使います。

さて、現代人が発明したコンピュータのことです。やはりゼロが出発点です。つまり、コンピュータを扱う世界でもゼロから数える癖があります。たとえば、一人目には0、二人目には1、三人目には2というふうに認識番号を割り振ります。また、n回何かをすることを「0回目からn-1回目まで」と表現します。artisocでも同様です。

もっとも、目の前に4人の友達がいるとき、0, 1, 2, 3と数えて、「この部屋に3人いる」などと言わないようにならう。だからといって、ゼロの存在を忘れてはいけません。たとえば、逆に1人ずつ部屋から出て行くことを想定すると、現状の4人から始まり、3, 2, 1, 0となります。1が最後ではありません。「部屋には誰もいない」という代わりに「部屋にゼロ人いる」というのは日本語としては変ですが、英語だと、There is no one in the room. のように言います。カウントダウンはまさにこのような数え方です。「いち」と叫ぶのと同時に花火に点火したりすると周囲の人は驚くに違いありません。

ところで気づいた人も多いと思いますが、この本では部や章や節はゼロから始まっています（ページについてはどうでしょうか？）。

実は序数の0, 1, 2, 3……と基数の0, 1, 2, 3……はちゃんと対応しているのです。それは、序数という集合の要素の数が基数になっているのです。まず、出発点は何もない虚無（nothing）です。何もない空くうの世界ですが、無が存在している（There is nothing: ネバーエンディング・ストーリーみたいですね）ともみなせます。そこでとりあえず空集合（ \emptyset ）を作りましょう。空集合が存在すれば、それを要素とする集合を考えられます。つまり $\{\emptyset\}$ （「何もない」ではなく、「何もないこと」）です。そうすれば、集合 $\{\emptyset, \{\emptyset\}\}$ も考えられます。だとすれば、 $\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$ も。この後、ずっと続いていきます。これが序数です。表記がどんどん煩わしくなるので、 \emptyset を0, $\{\emptyset\}$ を1, $\{\emptyset, \{\emptyset\}\}$ を2, $\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$ を3……と書くことにします。序数0, 1, 2, 3の要素の数は、たしかに0個、1個、2個、3個となっていますね。こうして序数に基数が対応していることがわかりました。めでたし、めでたし。

第29章

過去を参照する：アクセルロッドのゲーム戦略選手権モデル（簡略版）を作る

29.0 過去の履歴を現在の選択に結びつけます

- 繰り返し「囚人のジレンマ」ゲームの戦略選手権をモデル化します
- エージェントに過去に依存する行動ルールを持たせます
- GetHistory()関数を駆使します
- 「配列」になっている変数の扱いに慣れましょう

29.1 過去の状況を人工社会に組み込む

今まで作ってきたモデルでは、エージェントの相互作用や行動選択が、基本的にエージェントや周囲の環境の「現在の」状態、すなわち相互作用や行動選択のルールが実行されるステップにおける状態に依存していました。言い換えると、過去に周囲がどのような状況になっていたかといった情報は、エージェントの判断に影響を及ぼしませんでした。

しかし実際には、私たちの行動選択の多くは、過去の記憶の参照と表裏一体になっています。この章では、人工社会の過去と現在とをさまざまに結びつける技法を学びます。すでに第2部で森林火災やライフゲームのモデルを作った際に、「同期」の問題への対処法のひとつとして、直前の情報を知るためにGetHistory()という関数を利用しました。この関数をフォ文などこれまで学んできた基本的な技法と組み合わせて用いることで、過去に依存した現実的で複雑な行動を簡単にルール化することができます。

29.2 「囚人のジレンマ」ゲームと戦略の優劣

過去を参照する技法を学んでいく上で、格好のモデルがあります。それは、繰り返し「囚人のジレンマ」

ゲーム・モデルと呼ばれているものです。このモデルをもとにして、コンピュータ・シミュレーションによる戦略選手権を最初に試みたロバート・アクセルロッドの名前にちなんで、アクセルロッドの選手権モデルとも呼ばれています。

なお、アクセルロッドは選手権モデルを適者生存の原理に基づく適応モデルに発展させました。次章では、この適応モデルと、エージェントが優れた戦略を学習していくモデルとを作ります。

「囚人のジレンマ」とは、1950年代初めにアメリカで考案されたゲーム的な状況です。警察に逮捕された共犯容疑者2人にとって、2人が協力して否認（黙秘）すると互いに裏切って自白するより得になるが、相手が協力しようと裏切ろうと自分は裏切った方が得になる、という状況です。

囚人のジレンマは、下のようなゲームの構造として表すことができます。これを利得表と呼びます。表の各マスの左側の数値がプレイヤー1の利得を、右側の数値がプレイヤー2の利得を表しています。

プレイヤー1\プレイヤー2	協力 (C)	裏切り (D)
協力 (C)	3.0, 3.0	0.0, 5.0
裏切り (D)	5.0, 0.0	1.0, 1.0

「囚人のジレンマ」状況で、各人が合理的に判断すると「裏切り」を選択し、社会的に望ましい結果を産み出さないことがあります。それ以降、社会科学のさまざまな分野で、「囚人のジレンマ」状況が続く場合、どのようにしたら社会的に望ましい結果（つまり相互協力）が生じるのか、言い換えると高得点をもたらす戦略は何かという研究が行なわれ続けました。

そのような中で、1970年代末に政治学者のロバート・アクセルロッドが、コンピュータ・シミュレーションで戦略同士を競わせるコンテストを企画し、戦略のコンピュータ・プログラムを募集したのです。たくさんの応募があり、それらを競わせた結果、この問題を長年研究していたアナトール・ラバポートが応募してきたティット・フォ・タット（Tit for Tat略してTFT）という戦略（日本語では「目には目を」戦略とか「しっぺ返し」戦略とか訳されています）が優勝したのです。それは、最初は協力し、それ以降は相手の行動をそのまま

やり返す、という戦略でした。

アクセルロッドのシミュレーションをそのまま再現すると複雑になるので、ここでは簡略化したモデルを作ることにしましょう。

29.3 繰り返し「囚人のジレンマ」戦略コンテストの概要

具体的には、次のようなコンテストのモデルをこの章で作っていきます。

1. 図29.1のように配置された50のエージェントが上下ペアになって、「囚人のジレンマ」のゲームを100ステップの間繰り返しプレイします。

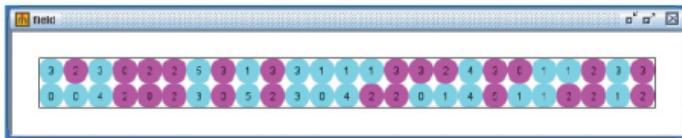


図29.1

2. プレイヤーは、各ステップ、ランダムに割り振られた行動ルール（戦略）にしたがって、相手と協力するか、裏切るかを決定します（これを「手」といいます）。とりうる戦略は、アクセルロッドのコンテストにもエントリーした、以下の6つの戦略です。モデルの中では、それぞれの戦略に0から5までの番号を振って、ルールの中で言及するときにはその番号を使います。

0. **Random 戦略** : ランダムに「協力」「裏切り」を選択する。
1. **AIC 戦略** : 相手の手に関わりなく常に「協力」を選択する。
2. **AIID 戦略** : 相手の手に関わりなく常に「裏切り」を選択する。
3. **TFT 戦略** : 最初は「協力」、以後は前回相手が「協力」なら「協力」、「裏切り」なら「裏切り」を選択する。
4. **Friedman 戦略** : 最初は「協力」、以後相手が裏切らない限り「協力」、一度でも裏切った

ら、それ以降全て「裏切り」を選択する。

5. **Tullock戦略**: 最初の10ステップは「協力」を選択する。以後、その間に相手が「協力」を選択した頻度よりも0.1低い確率で「協力」を選択し、それ以外は「裏切り」を選択する。
3. 各ステップ、プレイヤーの手の組み合わせに応じて、29.2で示した利得表の利得が各々に配分されます。利得はプレイヤー毎に累積加算されていきます。
4. 同時に、各プレイヤーの総利得を計算し、さらに戦略毎の平均利得を求め、戦略の優劣を比較します。

29.4 ルールの大枠

それでは、まずモデル作りの準備作業から取りかかりましょう。手順は以下の通りです。

1. Universeの下に空間fieldを追加します。大きさは 25×2 と横長の設定にします。あとはデフォルトのままで。
2. fieldの下にエージェントplayerを追加します（「エージェント数」は0のまま）。
3. playerには、戦略を示すstrategy（整数型）、「協力(0)」「裏切り(1)」の選択を表すmove（整数型）、累積の利得を納めるscore（実数型）、色を表すcolor（整数型）の4つの変数を追加します。色はプレイヤーが出手に連動して変化するものとします。このモデルでは、GetHistory()関数を使って、自分や相手の手を最大で99ステップ（ゲームの継続ステップ数-1）遡って参照することになります。そこで、第19章(19.5)の手順に従い、変数moveのプロパティで「記憶数」を99に設定しておいて下さい。
4. マップ出力の設定もしておきましょう。マップの表示範囲の最大値を24と1に変えます。要素設定でエージェントの表示色を「変数指定」にし、変数colorを指定しておきましょう。また、各プレイヤーがどの戦略をとっているのかわかるように、strategyを情報表示させて下さい（この技法は第23章(23.5)で学びました）。
5. このモデルをIPD（Iterated Prisoner's Dilemmaの略です）という名前で保存しましょう。

続いてplayerエージェントの配置と初期設定のルールを書き込みます。まずUniverseのルールエディタを開いて、Univ_Init{}のセクションに次のようなエージェントを50生成して配置するルールを書きます。

```
Univ_Init{
Dim i As Integer
Dim set As Agtset
For i = 0 To 49
    CreateAgt(Universe.field.player)
Next i
MakeAgtset(set, Universe.field.player)
RandomPutAgtsetCell(set, False)
}
```

続いてplayerのルールエディタを開き、エージェントにランダムに戦略を割り振るルールを書き込みます。Agt_Init{}のセクションに次のルールを書き込みましょう。

```
Agt_Init{
My.strategy = Int(Rnd() * 6) //0から5までの整数のどれか
}
```

29.5 6つの戦略をルール化する

各playerはY方向に隣接したplayerとペアになって「囚人のジレンマ」ゲームをプレイします。プレイの仕方は各自に割り振られた戦略に依存し、この戦略に従って、各ステップ「協力(0)」「裏切り(1)」いずれを選択するのかが決定されます。この部分をplayerのルールエディタにルールとして記述するのが次の作業です。

- 最初にAgt_Step{}の中身を示しておきましょう。6つの戦略のうち、TFT, Friedman, Tullockの3戦略についてはユーザ定義関数によって明記しています。ユーザ定義関数は、繰り返しの多い作業をまとめるだけでなく、複雑なルールを単純化(構造化)するためにも便利です。

```
Agt_Step{
Dim one As Agt
Dim set As AgtSet
```

```

//同じ列のプレイヤーを対戦相手とする
MakeAllAgtsetAroundPositionCell(set, Universe.field, My.X, 1
- My.Y, 0, 0)
one = GetAgt(set, 0)
//戦略に従って協調(0)か裏切り(1)かを決める
If My.strategy == 0 Then //Random戦略
    My.move = Round(Rnd())
Elseif My.strategy == 1 Then //AllC戦略
    My.move = 0
Elseif My.strategy == 2 Then //AllD戦略
    My.move = 1
Elseif My.strategy == 3 Then //TFT戦略(下のユーザ定義関数で明記)
    My.move = TFT(one)
Elseif My.strategy == 4 Then //Friedman戦略(下のユーザ定義関数で明記)
    My.move = Friedman(one)
Else
    My.move = Tullock(one) //Tullock戦略(下のユーザ定義関数で明記)
End if
//手に応じて変色
If My.move == 0 Then
    My.color = Color_Cyan
Else
    My.color = Color_Magenta
End if
}

```

そろそろ上書き保存しておいてください。ルールの全体的な構造はごく単純です。文法的には

```
MakeAllAgtsetAroundPositionCell(set, Universe.field, My.X, 1 -  
My.Y, 0, 0)
```

が新しい表現です。MakeAllAgtsetAroundOwnCell()が自分の周囲(AroundOwn)のエージェントをリストアップするのに対し、この関数は特定の位置の周囲(AroundPosition)のエージェントをリストアップする関数です。これを使って上か下いずれかに隣接するplayerをいったんエージェント集合型変数setにリストアップします。ここで、対戦相手の座標が自分の座標(My.X, My.Y)に対して(My.X, 1 - My.Y)と表せることに注意してください(Y座標は0か1しか取りませんので、Y方向に隣接したエージェントのY座標は、自分のY座標の値を反転することで得られます)。最後に0が二つ並んでいますが、最初はLayerの値(デフォルトなので0)で、次は視野の広さです。これが0なので、(My.X, 1 - My.Y)上に存在するエージェントしかリストアップしません。

2. ユーザ定義関数を使う複雑な3つの戦略のルールに移りましょう。いずれも、相手プレイヤーを引数、「協力(0)」か「裏切り(1)」を戻り値とする関数として記述します。まずTFT戦略です。

```
//1ステップ前の手をそのまま返すTFT戦略の定義  
Function TFT(person As Agt) As Integer{  
Dim action As Integer  
If GetCountStep() == 1 Then //ラウンド最初は協調  
    action = 0  
Else  
    action = GetHistory(person.move, 1) //それ以外は前期の相手の手  
    を返す  
End if  
Return(action)  
}
```

GetHistory(person.move, 1)とすることでエージェントperson(ルール実行時には上記oneが代入されます)の1ステップ前の手を参照することができます。あとはこの手をそのまま現在の自分の手とし

て、Return(action)で戻してやればいいだけです。

3. 次に、Friedman戦略のルール化です。

```
//過去一度でも裏切ったら裏切り続けるFriedman戦略の定義
Function Friedman(person As Agt) As Integer{
Dim i As Integer
Dim action As Integer
If GetCountStep() == 1 Then //最初は協調
    action = 0
Else //それ以外は相手の過去の裏切りをチェック
    action = 0
    For i = 0 To GetCountStep() - 2 //過去をさかのぼれるだけさかのぼる
        If GetHistory(person.move, i + 1) == 1 Then //裏切りが一度でもあれば裏切る
            action = 1
            Break //もう調べる必要がないからフォ文の外に出る
        End if
    Next i
End if
Return(action)
}
```

ここでのポイントは、フォ文とGetHistory()関数との組み合わせの部分です。

GetHistory(person.move, i + 1)という具合に、フォ文の変数と関連づけて、i + 1期前の相手の手を参照するようにしています。結局1ステップ前から「現在のステップ数 - 1」ステップ前(つまり1ステップ目)まで、相手の手全てを順次さかのぼってチェックすることになります。ただし、過去に一度でも相手が裏切れば自分の手が「裏切り」に確定するので、条件式がいったん満たされれば、それ以上過去

をさかのぼる必要がありません。そのためBreakがイフ文の中で使われています。

このように、GetHistory()関数を駆使して、過去のさまざまな情報を得ることができます。一点注意を要することがあります。それは、さかのぼるステップ数を指定する際に、実際の経過ステップ数を越えないようにすることです。GetCountStep()関数を使うなどして、経過ステップ数を越えて時間をさかのぼらないようにする工夫が必要になります。

4. 最後に、Tullock戦略は以下のようないくつかの関数になります。

```
//開始10ステップの相手の対応で手の出し方を変えるTullock戦略の定義
Function Tullock(person As Agt) As Integer{
    Dim i As Integer
    Dim action As Integer
    Dim num As Double
    Dim rate As Double
    If GetCountStep() <= 10 Then      // 最初の10ステップは協力
        action = 0
    Else    //11ステップ以降
        //相手の最初の10ステップの協力頻度をチェック
        num = 0
        For i = 0 To 9
            If GetHistory(person.move, GetCountStep() - 1 - i) == 0
        Then
            num = num + 1
        End if
    Next i
    //これに基づき自分の協力確率を計算し手を決定
    rate = num / 10 - 0.1
```

```

If rate < 0 Then
    rate = 0
End if
If Rnd() < rate Then
    action = 0
Else
    action = 1
End if
End if
Return(action)
}

```

特に新しい技法は使われていません。GetHistory(person.move, GetCountStep() - 1 - i)することで、1+ iステップ目に相手がとった手をチェックしていることはわかりますね。

5. 100回ゲームが繰り返されたら対戦が終わるルールです。そこで、ステップ数100(101ステップ目)になつたらシミュレーションが終了するように、Univ_Step_End{ }の部分に、次のルールも書いておきましょう。

```

Univ_Step_End{
//ゲームの終了
If GetCountStep() == 100 Then
    ExitSimulation()
End if
}

```

artisocでは、ステップ数0(1ステップ目)でUniv_Init{ }とAgt_Init{ }が実行されるので、ステップ数1(2ステップ目)から対戦が始まります。したがって100回目の対戦はステップ数100(101ステップ目)です。

ここまで書き上がったら、上書き保存してモデルを実行してみましょう。playerが各々の戦略にしたがって囚人のジレンマゲームをプレイし始めるはずです。

29.6 利得を集計する

各ステップにおける「協力」と「裏切り」の手の組み合わせに応じて、playerに利得を割り当てれば、スコアの集計は完成します。手の組み合わせと利得との対応関係は、29.2の利得行列に示したとおりです。自分がC、相手がDのときは利得0.0、自分がD、相手がCのときは利得5.0といった具合に、それぞれの手の組み合わせに対する二人のプレイヤーの利得が全く同じであること（これを「利得行列が対称である」と言います）に注目してください。このことを利用して、以下では各プレイヤーの利得行列を二次元配列（第25章を参照）で表し、利得を計算する際にルールの中で活用することにします。

まず、Universe直下に利得行列を表す 2×2 の二次元配列変数pmatrixを追加します。「次元数」は「2」、「1次元目の配列数」「2次元目の配列数」はともに「2」にします。変数型は実数型にしておいて下さい。その上でこの変数の初期値設定をしましょう。

今まででは、変数の初期値はルールエディタの中で設定するかデフォルト値をそのまま使ってきました。しかしartisocでは、**設定**メニューから**初期値設定**を開いて値を与えることも可能です（[図29.2](#)参照）。

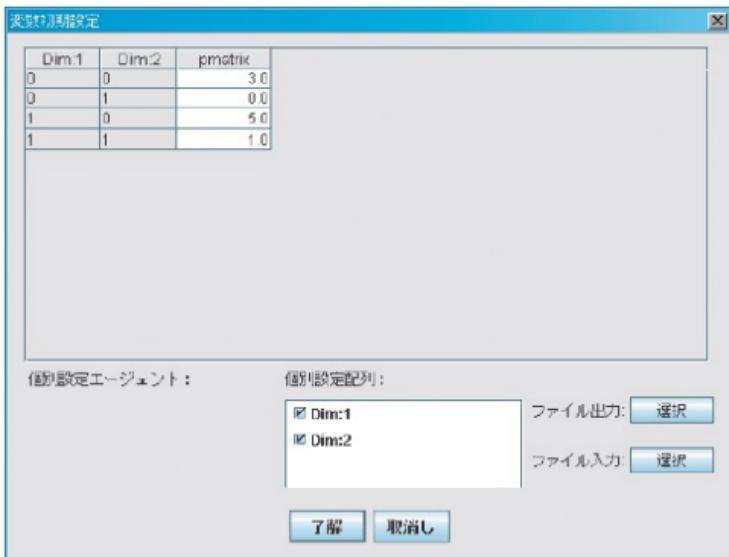


図29.2

このやり方で、 $\text{pmatrix}(0, 0) = 3.0$, $\text{pmatrix}(0, 1) = 0.0$, $\text{pmatrix}(1, 0) = 5.0$, $\text{pmatrix}(1, 1) = 1.0$ というふうに配列の各要素に利得の値を割り振りましょう。つまり、この配列変数`pmatrix`の一次元目の要素番号(0か1)は自分の手を、二次元目の番号は相手の手を表していると解釈するわけです。たとえば、 $\text{pmatrix}(0, 1)$ には、自分がC、相手がDの場合の自分の利得0.0が納められています。

IPD(B)という名前を付けて保存して下さい。

ここまで準備をすると、利得集計のルールは簡単に書けます。Universeのルールエディタを開き、`Univ_Step_End{ }`の中の、シミュレーションの終了に関するルールの前の部分に、以下のようなルールを書き加えましょう。

```
Univ Step_End{
//利得集計
Dim i As Integer
```

```

Dim p1 As Agt
Dim p2 As Agt
Dim set As Agtset
For i = 0 To 24 //25ペアの全てについて
    //上下段のplayerを取り出す
    MakeAllAgtsetAroundPositionCell(set, Universe.field, i, 0, 0,
0)
    p1 = GetAgt(set, 0)
    MakeAllAgtsetAroundPositionCell(set, Universe.field, i, 1, 0,
0)
    p2 = GetAgt(set, 0)
    //利得行列を使ってスコア(累積)を割り当てる
    p1.score = p1.score + Universe.pmatrix(p1.move, p2.move)
    p2.score = p2.score + Universe.pmatrix(p2.move, p1.move)
Next i
//ゲームの終了

```

フォ文を使って、空間左端から順次playerのペア(p1とp2)をチェックしていき、両者の「協力」「裏切り」の組み合わせ(p1.moveとp2.move)から、利得行列pmatrixを参照し、それぞれが受け取る利得を各playerのscore変数に加えていきます。

上書き保存しておきましょう。

29.7 戰略のパフォーマンスを比較する

最後に、6つの戦略を取るplayerたちが1ステップ、1エージェントあたり平均どれだけの利得を上げているのかを戦略ごとに出力させましょう。出力形式は「棒グラフ」にします。棒グラフによる出力設定は初出なので、作業手順をていねいに説明しておきます。

- まずUniverse直下に各戦略の平均利得の出力のための変数を作ります。ここでは、出力値が6つに及ぶので、一次元配列を使って一括して管理するのが便利です。そこで、Universeに実数型の変数avgpayoffを追加し、プロパティで「次元数」を「1」に、「1次元目の配列数」を「6」に設定しましょう。配列変数avgpayoffの要素番号0～5は、6つの戦略の番号に対応していると考えます。たとえば、avgpayoff(3)には、TFT戦略の平均スコアが代入されることになります。
- 続いて、avgpayoffの6つの要素を棒グラフに出力させるための設定をしましょ

う。出力設定 の 出力項目リスト において、追加する出力種類 プルダウンから 棒グラフ を選んで 追加 をクリックします。「棒グラフ設定」ダイアログで、グラフ名等を入力した上で、「棒グラフ要素リスト」に順次6つの出力要素を追加していきます。要素設定は、たとえばTFT戦略の場合、「出力値」の部分に「Universe.avgpayoff(3)」と入力することになります。「出力値表示」のチェックボックスをオンにすると、棒グラフと一緒に変数値も出力することができます。なお、小数表示をデフォルトの0桁ではなく1桁にして、Y軸の最大値もデフォルトの100ではなく10か5にしておいたほうが見栄えがよくなります(図29.3参照)。



図29.3

- 最後に、戦略毎の平均スコアを算出するためのルールを書けば完成です。各戦略について、総スコアと、戦略を採用しているplayerの総数を集計した上で、前者を後者で割り、さらにステップ数で割

れば平均スコアが求まります。配列変数avgpayoffの要素番号が、playerの変数strategyで指定される戦略番号に対応していることに留意すると、下記のようなルールが書けます。

Univ_Step_End{ }の利得割り当てのルールに続けて書いていくといいでしよう。ただし一時的変数の変数型宣言文は、冒頭に書き込んで下さい。

```
//セクション冒頭に書き込む

Dim strt As Integer
Dim one As Agt
Dim sum(6) As Double //1次元、配列数6の実数型
Dim num(6) As Double

//各戦略の平均スコアを計算
For i = 0 To 5 //配列の初期化
    sum(i) = 0
    num(i) = 0
Next i
//全playerについて、戦略毎の総利得と人数をカウント
MakeAgtset(set, Universe.field.player)
For each one in set
    strt = one.strategy //戦略を調べる
    //当該戦略のスコアと人数を加算
    sum(strt) = sum(strt) + one.score
    num(strt) = num(strt) + 1
Next one
//最後に頭数×ステップ数で割ってスコアの平均値を算出
For i = 0 To 5
    If num(i) == 0 Then //全く選ばれていない戦略について
        Universe.avgpayoff(i) = 0
```

```

Else
    Universe.avgpayoff(i) = sum(i) / (num(i) *
GetCountStep())
End if
Next i

```

上書き保存をした上で、実行ボタンを押してみましょう。6本の棒グラフが並んで出力されれば成功です。どの戦略が高いスコアをあげるでしょうか。

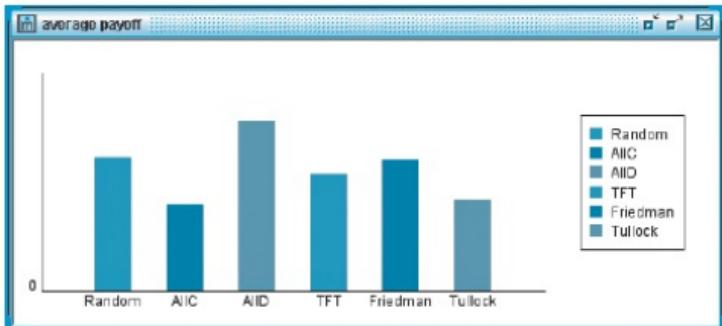


図29.4

新しく学んだ事項

- MakeAllAgtsetAroundPositionCell()
- GetHistory()を駆使して、過去の情報をエージェントが入手する
- 設定メニューの「初期値設定」の利用
- 棒グラフの設定方法
- ユーザ定義関数をルールを見やすくするために使う

練習問題

最初の2つは、同じルールをもっと簡単に書くポイントを見つける練習(頭の体操)です。最後の問題は、過去を参照するルールを書く練習だけでなく、自分でユーザ定義関数(Function)を作る練習も兼ねています。

29.1

相手が一度でも裏切れば、以後裏切り続けるFriedman戦略だが、上のルールは定義を忠実にそったものだ。実は、過去の相手の全ての手をチェックしなくても、ルールとしてもっと簡単に記述することができる。そこで、頭の体操です。Friedman戦略のルールをもっとスリムなものにしてみよう。



過去一期分の情報があれば十分です。

29.2

Tullock戦略では、11ステップ目以降は毎ステップ、同じ計算を繰り返しているという無駄をしている。実は、この戦略だけのためにエージェントに変数を追加すると、この無駄を省くことができる。そうしてから、Tullock戦略をもっと簡潔なルールにしてみよう。

29.3

IPDモデルに第七の戦略を登場させよう。TFT戦略に少し嫌らしさを加え、相手の手をただ返すだけでなく、相手が二度続けて「協力」を選択した場合、付け込んで「裏切り」を選択するようにしてみる。名称は2CforD戦略とでもしておこう。



経過ステップ数を越えて時間をさかのぼらないように注意しましょう。

繰り返し文いろいろ

「フォ文」は、ある所定の回数分ルールを繰り返し実行するための構文でした。たとえば、

```
For i = 0 To Universe.ninzu-1  
    CreateAgt(Universe.hiroba.hito)  
Next i
```

とすることで、変数ninzuの数だけCreateAgt()関数を実行し、指定した数のエージェントを生成することができました。

これに対して、あらかじめ回数を定めない繰り返しも考えることができます。たとえば、一様乱数発生関数Rndを使って「0.5より大きく0.7より小さい」でたらめな数を発生させる方法を考えてみましょう。幾つか方法はありますが、一番ストレートなやり方は、条件が満たされるまで繰り返しRndを実行する方法でしょう。この場合、Rndを何回繰り返すことになるのか、あらかじめ知ることはできません。

artisocでは、このような不定回数の繰り返しを、以下のような構文で表現することができます。

```
Do While 条件式  
    ルール  
Loop
```

```
Do Until 条件式  
    ルール  
Loop
```

上(Do While文と言います)は条件式が満たされている間、下(Do Until文と言います)は条件式が満たされるまで、指定したルールを繰り返し実行する構文です。たとえば、先ほどの乱数発生の繰り返し処

理は、Do Until文を使って次のように書き表すことができます。

```
r = Rnd( )  
Do Until r > 0.5 And r < 0.7  
    r = Rnd( )  
Loop
```

ただし、これらのDo繰り返し構文を使用する際には、細心の注意が必要です。というのも、これらの構文では、条件式の設定次第で、artisocが繰り返しのループから抜け出せなくなり、フリーズしてしまう「無限ループ」が発生するからです。たとえば、上の例で条件式を「r > 1.0」などと間違えてしまうと、Rndは1.0未満なので、たちまち無限ループが生じてしまいます。ルールが複雑な場合、無限ループの危険性はさらに高まります。たいていの繰り返し処理は、フォ文で書くことが可能ですが。本文中でDo While文・Do Until文に触れていないのも、こうした事情からです。

(阪)

第30章

エージェントに学習・適応させる：ゲーム戦略選手権モデルを発展させる

30.0 エージェントを賢くする第一歩です

- エージェントがダイナミックに行動ルールを変化させます
- 「模倣による学習」モデルを学びます
- 「学習」の導入に伴いシミュレーションの時間が重層化します
- 順序づけに基づいた「適者生存」のルール化も学びます
- 最後に、アクセルロッドの適応モデルの簡略版を作ります

30.1 学習・適応エージェントへ「はじめの一歩」

前章の「囚人のジレンマ」モデルでは、playerエージェントの戦略が固定されていました。アクセルロッドは、高得点をもたらす戦略が優勢になっていく（低得点しかもたらさない戦略は淘汰されていく）過程についても、さらにモデル化しています。そこでこの章では、自分のスコアだけでなく周囲のエージェントのスコアに依存して自分の戦略が変化するようにモデルを修正しましょう。つまり、環境からのフィードバックを受けて自らの行動パターンをダイナミックに変えるモデルの初步を学びます。

最初に、「模倣による学習」をモデル化します。エージェントが周りを見渡して、優れた戦略を自分も採用します。次に、エージェントの行動パターンの変化は、環境への「適応」としても理解することができます。本章では、「適者生存」のメカニズムをモデル化する、初步的な技法も学びます。アクセルロッドの適応モデルほど複雑なルールではありませんが、基本的な考え方は同じです。

さて、このような「学習・適応」をモデル化するには、「時間」についての注意が必要です。学習については、毎ステップ、エージェントが少しづつ学習して賢くなるルールも可能でしょう。しかし、少なくとも適応過

程については、個体が生きている時間と世代交代していく時間とを区別する必要があります。この章では、相互作用している普通のフェーズと、学習・適応する特別のフェーズとを区別する技法を学ぶことにしましょう。

30.2 繰り返し「囚人のジレンマ」モデルの時間の重層化

まず、エージェントが学習する過程をモデル化しましょう。繰り返し「囚人のジレンマ」をプレイする上で、学習のポイントは、自分がとってきた戦略より優秀な（高得点をもたらす）戦略が見つかると、優秀な方を選び取る、という点にあります。そこで、次のようにモデルを複雑にしましょう。

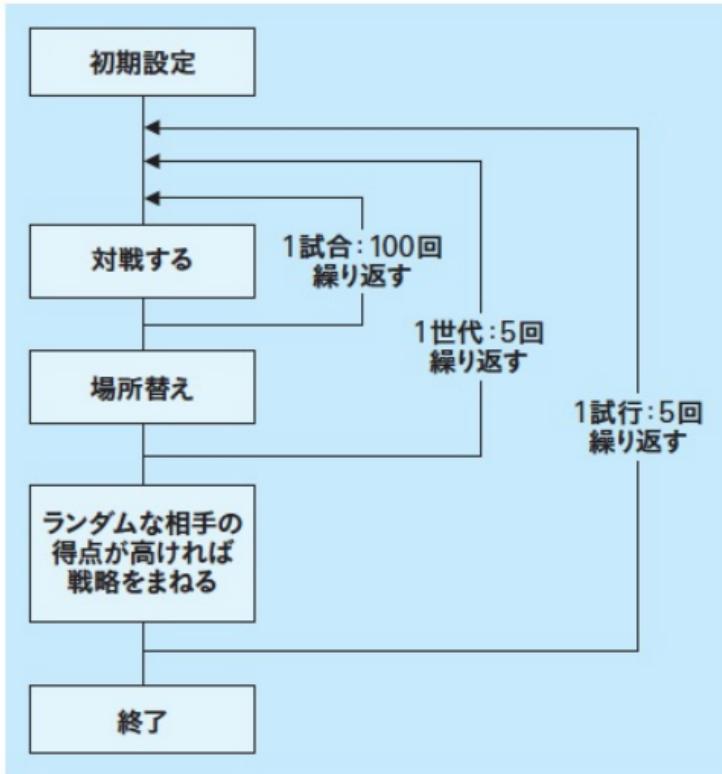


図30.1

- 各エージェントは、同一相手と100回対戦する（前章と同じ繰り返し「囚人のジレンマ」ゲームをする、これを1試合と呼ぶ）
- 同じ戦略のままで、5エージェントを相手に（つまり、いろいろな戦略を相手に）繰り返しゲームを1試合ずつする（これを1世代と呼ぶ）
- 最後に、各エージェントはランダムに選ばれた他のエージェントと総得点を比較し、相手の方が高ければ、次の世代から相手がとっていた戦略を用いる（エージェントが学習する）

4. 以上を5世代繰り返す

これから作る新しいモデルは、前章で作ったモデルIPD(B)をいったん開いて、それを新たにIPD(C)という名前を付けて保存して下さい。これから、IPD(C)を用います。エージェントが学習するモデルでは、シミュレーションの時間の流れが重層化することがわかると思います。シミュレーションのステップ数は自動的に増えています。それを適当なルールで、対戦数、試合数、世代数に変換する必要があります。そこで、Universeに新しい3つの整数型変数、nomove, nogame, nogenerationを追加して下さい。なお、時間の重層化をめぐる技法については、第4部(第31章)で体系的に説明しています。

まずUniverseのルールエディタの中を書き換えます。まず、Univ_Init{ }のエージェントを生成するルールはそのまま残しておいて、エージェントを配置するルールを削除して下さい。エージェントの配置は試合毎に変わるので、Univ_Step_Begin{ }にあらためて書き直します。

続いてUniv_Step_Begin{ }に下記のようなルールを書き込みます。ステップをもとにして、実行順序をコントロールするための3つの変数に値を与え、試合ごとにエージェントの「席替え」=対戦相手の変更を行ないます。

```
Univ_Step_Begin{
Dim set As Agtset
//時間の管理
Universe.nomove= (GetCountStep() - 1) Mod 100 + 1 //対戦数
Universe.nogame = ((GetCountStep() - 1) \100) Mod 4 + 1 //試合数
Universe.nogeneration = (GetCountStep() - 1) \400 + 1 //世代数
//各試合のはじめに「席替え」して対戦相手を交代
If Universe.nomove == 1 Then
  MakeAgtset(set, Universe.field.player)
  RandomPutAgtsetCell(set, False)
End if
}
```

ここで商を整数值で求める割り算(\)をしていることに注意して下さい(コラム「[割り算こわい](#)」参照)。

次に、Univ_Step_End{ }の最後に書かれているシミュレーション終了のルールを変えます。具体的には、If GetCountStep() == 100 Thenの部分を

```
If Universe.nogeneration == 5 And Universe.nogame == 4 And  
Universe.nomove == 100 Then
```

とします。これで、5世代経ったら、シミュレーション全体が終了します。また、戦略毎の成績を計算している部分にもGetCountStep()が使われていますが、これをUniverse.nomoveに変えて下さい。

最後に、エージェントのルールエディタの中で、ステップ数に依存していたルールを、新たに導入した対戦数(nomove)に依存するように、中身を書き換えます。具体的な手順は、artisocのルールエディタにある「置換機能」を使って、「GetCountStep()」を「Universe.nomove」に一括置換するだけです(図30.2参照)。



図30.2

30.3 「模倣による学習」のルール化

各世代最終時に、自分の得点より他のエージェントの得点が高い場合、そのエージェントの戦略を模

倣するルールを導入します。本来はエージェントの行動ルールなのですが、Univ_Step_End{ }でスコアを集計したあのルールなので、ルールを書く場所もUniv_Step_End{ }の部分のスコア集計ルールのあと、終了条件のまえの箇所です。

「学習」の効果は、世代が変わったあの最初の試合のときにいっせいに現れるので、同期を取る必要があります。そこでplayerに、次にくる戦略のバックアップ変数nextstrategy(整数型)を追加しましょう。

Univ_Step_End{ }には、次のようなルールを書き加えます。変数の型宣言以外は、必ず、戦略の平均スコアを計算するあの部分に書くようにして下さい。

```
Univ_Step_End{
Dim cand As Agtset //冒頭に宣言
```

(中略)

```
If Universe.nogame == 4 And Universe.nomove == 100 Then
//世代交代
  //各playerは優れた戦略を模倣して「学習」
  For each p1 in set
    //自分以外のplayerを無作為抽出
    DuplicateAgtset(cand, set)
    RemoveAgt(cand, p1)
    p2 = GetAgt(cand, Int(CountAgtset(cand) * Rnd()))
    //相手のスコアが自分のを上回れば相手の戦略を真似る
    If p2.score > p1.score Then
      p1.nextstrategy = p2.strategy
    Else
      p1.nextstrategy = p1.strategy
    End if
  Next p1
  //次世代に入る前に、戦略を更新しスコアを初期化
  For each p1 in set
    p1.strategy = p1.nextstrategy
    p1.score = 0
  Next p1
End if
```

これで学習させるルールが完了します。必ず自分以外のエージェントと比較させるために、すこし技巧的なルール(DuplicateAgtset()とRemoveAgt()の組み合わせ)を書いてあります。

それでは、上書き保存しておきましょう。

30.4 戦略の優劣を視覚化する

エージェントが学習をするようになると、戦略はエージェントがランダムに選択している結果ではなくなります。たくさんのエージェントが採用している戦略もあれば、ほとんど使われなくなる戦略もあるでしょう。そこで、各戦略の平均スコアとは別に、各戦略を何人のplayerが採用しているのかを、棒グラフで出力することにしましょう。新たな整数型の一次元配列変数strategy_distr(要素数は戦略の数だけ)をUniverseに追加して、棒グラフの出力設定をしましょう。以下では戦略数6の場合を想定したルールが書かれています。前章の練習問題29.3(2CforDの追加)をモデルに反映させているなら戦略数7でルールを書いて下さい。

各戦略の人数は、すでに前章29.7で、Univ_Step_End{ }の一時的変数numに集計しています。基本的にこれをUniverse.strategy_distrに置き換えるだけです。出力値の集計ルールは下記のとおりです。

```
Univ_Step_End{
(前略)

//各戦略の平均スコアを計算
For i = 0 To 5    // 配列の初期化
  sum(i) = 0
  Universe.strategydistr(i) = 0
Next i
//全playerについて戦略毎の総利得と人数をカウント
MakeAgtset(set, Universe.field.player)
For each one in set
  strt = one.strategy //戦略を参照
  //当該戦略のスコアと人数を加算
  sum(strt) = sum(strt) + one.score
  Universe.strategydistr(strt) = Universe.strategydistr(strt) +
1
Next one

//最後に頭数×当該世代の経過対戦数で割ってスコアの平均値を算出
For i = 0 To 5
  If Universe.strategydistr(i) == 0 Then //全く選ばれていない戦略について
    Universe.avgpayoff(i) = 0
  Else
    Universe.avgpayoff(i) = (sum(i) / Universe.strategydistr(i))
```

```
/ (100 * (Universe.nogame - 1) + Universe.nomove)
End if
Next i
}
```

これで完成です。上書き保存してから実行しましょう。

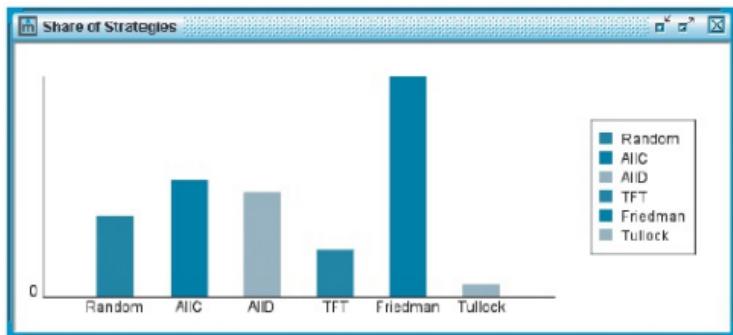


図30.3

30.5 「適者生存」のルール化

30.3では、個々のエージェントが学習することにより、優秀な戦略をとるエージェントが増えていく過程をモデル化しました。ここで、発想を変えて、世代交代時には全体を見渡して、高いスコアを上げたエージェントのみが選別され、低いスコアのエージェントは淘汰されてしまうルールにしてみましょう。つまり、学習ルールに代えて、「適者生存」の初步的なルールを導入します。

具体的には、playerエージェントをスコアに基づき降順（高い方から順次低くなる）でソートします。エージェントがある変数の値にしたがってソートする場合に使う関数がSortAgtset()です。その上で、上位20%のエージェントが同じ戦略を持つ「クローン」を4人ずつ生成するルールにします。つまり、上位の10エージェントが40エージェント生成するので、世代交代時には、50エージェント存在しています。エージェント数を一定にするために、下位80%（つまり40エージェント）を消すルールにします。

先ほど学習のルールを書いたUniv_Step_End{ }の部分で、学習ルールを削除し、その代わりに下記のようなルールを書いていって下さい。

```
Dim j As Integer //冒頭に宣言

//適者生存のルール
If Universe.nogame == 4 And Universe.nomove == 100 Then //「世代交代」
    SortAgtset(set, "score", False) //playerをスコアの降順にソート
    //上位10エージェントを残してコピーを生成し、残りを削除
    For i = 0 To 49
        p1 = GetAgt(set, i)
        If i <= 9 Then //上位10位以内の場合
            For j = 0 To 3
                p2 = CreateAgt(Universe.field.player)
                p2.strategy = p1.strategy
            Next j
        Else //それ以外の場合
            KillAgt(p1)
        End If
    Next i
    //次世代に入る前に、スコアを初期化
    For each p1 in set
        p1.score = 0
    Next p1
End if
```

上のルールの中のSortAgtset(set, "score", False)は、エージェント集合setに納められているエージェント(具体的にはUniverse.field.player)の全てについて、Universe.field.player.scoreの降順(False)にソートさせる関数です。FalseをTrueに変えると、昇順にソートされます。ここで、ソートする基準の変数が文字列型("score")になっていることに注意して下さい。複数のエージェント種を1つのエージェント集合に入れてソートできるようにこのような仕様になっています。

適者生存のルールは、上のようにエージェント総数が一定という条件下では、上位エージェント(戦略)を増やす数と同じ数だけ下位エージェント(戦略)を減らすように気をつけましょう。

この修正版は、IPD(D)という名前で保存して下さい。

IPD(C)とIPD(D)とでは、優秀な戦略が広がっていくプロセスに何か違いがあるでしょうか。違いは明白

ではないかも知れませんが、モデルの考え方には大きな違いがあります。学習の場合は、エージェント自身が戦略を取捨選択しています。適応の場合は、全体状況を見渡した上で戦略が取捨選択され、エージェントのとる戦略に反映されます。

新しく学んだ事項

- エージェントに学習させる
- エージェントの行動を適応(淘汰)させる
- エージェントをソートする関数SortAgtset()
- ステップ数を基礎に時間の流れを重層化する
- ルールエディタの置換(検索)機能

練習問題

学習や淘汰の現象は、いろいろなルールで表現できます。柔軟に考えるクセをつけましょう。

30.1

学習ルールを変えてみよう。

各エージェントが世代交代時に無作為にエージェントを5人選んで、最もスコアの高いエージェントの戦略を模倣する。

30.2

淘汰ルールを変えてみよう。

世代交代時に、低得点10エージェントが消える一方、高得点10エージェントが自分と同じ戦略

をとるエージェントを1体産む。

割り算こわい

四則演算のなかで割り算は少し変わった性質を持ちます。整数と整数を足しても整数、引いても整数、かけても整数というふうに、他の演算は、整数同士の演算の結果は整数にしかなりません。いわば整数の世界で閉じています。しかし、割り算は、整数を整数で割っても答(商)が整数とは限りません。そのためartisocで割り算をするときは、ちょっと注意をしておく必要があります。

まず、割り算には2種類あります。商を実数値として求める除算(/)と商を整数値として求める除算(\)です。前者(/)では、除算の結果は実数値として与えられます。被除数(実, dividend)や除数(法, divisor)が整数でも実数でも、計算結果は実数となります。それに対して、後者(\)では、除算の結果は整数部分のみが与えられます。被除数(実, dividend)や除数(法, divisor)が整数でも実数でも、計算結果は整数値となります。小学校のころに学んだ剩余のある割り算ですね。

ちなみに、\はパソコンによっては¥と表示されます。しかし、artisoc(というよりコンピュータは全て)にとっては、実はこの二つの文字は同じものとして認識されます。どちらを使っていても問題はないので心配はご無用です。

$$10 / 3 = 3.3333\dots \quad 10 \backslash 3 = 3$$

$$10 / 3.0 = 3.3333\dots \quad 10 \backslash 3.0 = 3$$

$$10.0 / 3 = 3.3333\dots \quad 10.0 \backslash 3 = 3$$

$$10.0 / 3.0 = 3.3333\dots \quad 10.0 \backslash 3.0 = 3$$

あまり

「剩余」を求めるには、Modを用います。この演算は実行順序の管理で重要な働きをします。これは第30章や第31章で学びます。

$$10 \bmod 3 = 1$$

$$10.5 \bmod 3 = 1.5$$

10.5 Mod 2.5 = 0.5

また、計算結果をしまう変数の型にも注意しておきましょう。計算結果が実数であっても、整数型の変数におさめると、自動的に（親切にも）整数型に変換されてしまいます。その際、小数点以下の値は切り捨てになります（3.6666...は3になります）。逆に、整数として求めた値を実数型の変数におさめると、やはり自動的に実数型に変換されてしまいます。この場合、値は変わりません（3は3.0になります）。

割り算を使用するうえでの注意としては、0で除算しないように気をつけることも重要です。もっともこの場合は、すぐにエラーが出てシミュレーションが止まるので、深刻なミスにはなりにくいでしよう。

（光）

第4部

研究・実務のツールにする

いよいよこれからは、研究や実務のために、皆さんのが自分自身でオリジナルな人工社会モデルを作る番です。第3部までで学んださまざまな技法を適宜組み合わせて、作りましょう。しかし、モデルが動けば成功というわけにはいきません。単に動くだけでなく、研究や仕事の役に立つモデル、他の人を納得させるモデル、あるいは感動させるモデルを作らなくてはなりません。シミュレーションの分析も大事です。この第4部では、人工社会モデルを自分自身のモデルとして、自信の持てるものにしたり、洗練されたものにしたりするartisocの技法を学びます。また、artisocのモデル・シミュレーションとモデルの外の世界とのインターフェースの取り方についても考えてみます。そして最後に、技法から離れて、人工社会の考え方についても思いをめぐらせます。

第31章 実行順序を制御する

31.0 時の流れに流されないように

- ステップ内の実行順序を指定できます
- 注意しないと思わぬ失敗の原因になります
- `Agt_Init{ }`や`Univ_Finish{ }`などが実行されるタイミングを知りましょう
- 複数ステップで擬似的な1ステップを作り複雑な相互作用を表現します

31.1 実行順序は意外な盲点です

第19章で、山火事の例を使って、エージェントの行動を同期させることの重要性を学びました。同期をめぐる問題は、もう少し一般的に考えると、シミュレーションを実行している最中にどのような順序でエージェントに行動させるのかという問題に含めることができます。この「実行順序」は非常に重要です。なぜなら、同一エージェント種の中で、あるいは異なるエージェント種の間で、複数のエージェントをどのような順番で行動させるのかによって、シミュレーションの結果が変わってしまうことがあるからです。

この実行順序の問題は、ステップ内の実行順序とステップをまたいだ実行順序の二つに分けて考えることができます。

31.2 ステップ内の実行順序

既に、1ステップの一瞬の間に実は長い時間がある、ということを学びました。その「長い時間(=1ステップ)」の中身について、もう少し詳しく見てみましょう。ふつう1ステップは、`Univ_Step_Begin{ }`, `Agt_Step{ }`, `Univ_Step_End{ }`、から成っています(そうでないステップ—最初と最後—については次節で解説します)。エージェントが3つあれば、`Univ_Step_Begin{ }`, `Agt_Step{ }`, `Agt_Step{ }`, `Agt_Step{ }`

}, Univ_Step_End{ }といった具合にAgt_Step{ }が合計で3回実行されます。

ここで問題になるのはその順番です。デフォルトのままモデルを実行すると、毎ステップ、エージェントを全てごちゃ混ぜにし、順番をランダムに入れ替えて実行します。このランダムの順番によって、森林が燃え尽きたり、燃え残ってしまう例を第19章で見ましたね。そのときには、同期をとて解決する方法を紹介しましたが、モデルによっては同期を取らない方がよい場合もあります。たとえば、空き地が1つしかない分居モデルを考えてみて下さい。全てのエージェントが同時に判断したら、ひとつの空き地に、沢山の不満なエージェントが殺到してしまいます。現実でも大家さんが全く同時に入居を申し込まれる事は希でしょうから、やはり、ひとつのステップの中にも順番があった方が便利な場合もあるのです。

artisocでは、ステップ内での実行順序は、**設定** > **実行環境設定** の中の **実行順序** タブ(ダイアログボックスの上部にあるタブ)で、細かく設定することができます。それぞれの設定の特徴を見てみましょう。

◎ランダム

デフォルトの状態はランダムです。エージェント種に関わらず、毎回全てをシャッフルしてランダムに実行します。分居モデルのように、エージェント種(dimとpennyなど)による区別なしに、ごちゃ混ぜにして実行したい場合に適しています。また、同期をとる場合にも、普通この設定のままにしておけばよいでしょう。

◎エージェント種別指定

一方、エージェント種別指定を選ぶと、より細かく、エージェント種ごとの実行順序を指定する画面が現れます。同じエージェント種の内部では毎回シャッフルされます。左端の数字が実行順序を表します。エージェント種をクリックした状態で、右側の や のボタンを押す事で、この数字を変える事ができます。たとえば、[図31.1](#)のように設定すると、carとtruckをごちゃ混ぜにして実行し、その後、speedcamera(速度取り締まりカメラ)を実行します。こうすれば、carやtruckが速度を変化させた後に、そのステップでのスピードをチェックするモデルを作ることができます。

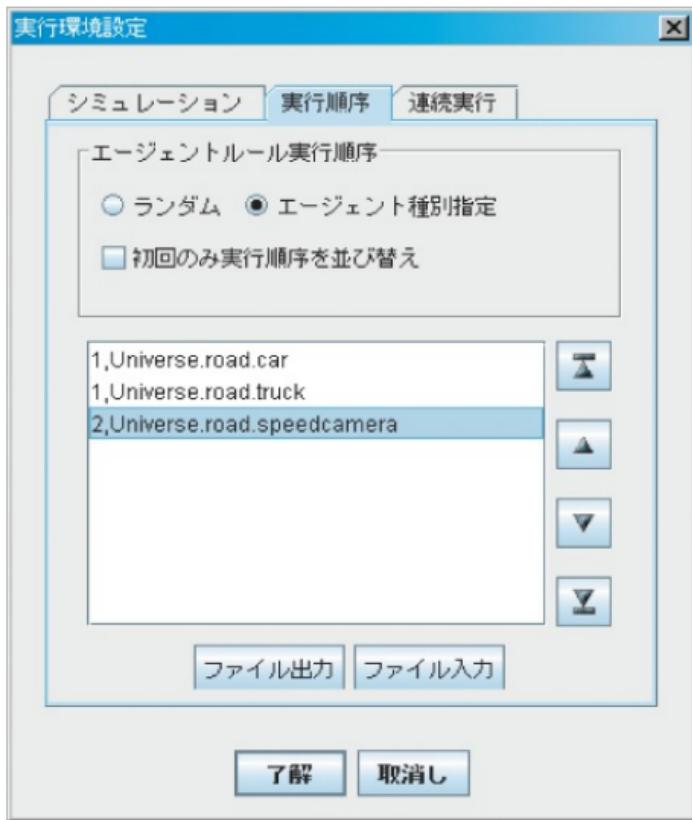


図31.1

このように、特定のエージェントを、他のエージェントが行動した後（もしくは前）に行動させたい場合には、エージェント種別指定が便利です。実行順序は慎重に選ばないと思わぬ失敗を招きます。どんな失敗が起こりうるのか、代表的な例を見てみましょう。

- 〈失敗例A〉時間の空白：この速度取り締まり機の例でランダムな実行順序を選んでしまうとどうなるでしょう？ 取り締まり機より先に実行されたクルマについては現在のステップの速度を調

べられます。しかし、残りのクルマはそのステップの行動をしていないわけですから、前ステップの速度しか調べられません。その後に違反してもそのステップでは捕捉できません。もし、この違反車両が次ステップで取り締まり機が行動する前に減速したら、取り締まり機は目の前の違反を見逃してしまいます。

- 「失敗例B)同種の連続実行による影響」逆に、実行順序を指定する事による失敗もあります。分居モデルで実行順序を

1. Universe.chessboard.dim e
2. Universe.chessboard.penny

とするとどういうことが起きるでしょうか?こうすると、dim e全てが連続して行動し、その後にpennyが行動します。dim eが移動した後の空き地は、dim eにとって不満の多い場所ですから、次のdim eがそこに移動しても、次のステップにはやはり不満に感じて出していく可能性が高くなります。逆に、もし pennyが移動してきていたら、そこに満足する可能性が高いので、dim eが不満に思う土地が空き地でなくなります。ですから、同じ色のコインが連続して実行するかどうかで、均衡状態になるまでの時間や、場合によっては結果にさえ、影響が出ないとは言い切れないのです。

◎「初回のみ実行順序を並び替え」オプション

ランダムを選択しているときに、このチェックを入れると、エージェント種に関わらず、初回のみ全てをシャッフルします。エージェント種別指定を選択しているときには、同じ番号の付いたエージェント種内部をシャッフルします。2ステップ目からは初回と同じ順序で実行します。

なぜこのようなオプションが必要なのでしょうか。狭い路地で警察が泥棒を追いかけるモデルを作って説明しましょう。

1. 15×1 の空間alleyを作成し、policeとthiefエージェントをそれぞれ一体ずつ作って下さい。
2. 出力設定して下さい。空間表示では、黒線を表示しY軸の最大値を0にしておくと、それらしく

見えます。

ルールは簡単です。エージェントは右に向かって走るだけです。thiefは

```
Agt_Init{
My.X = 5
My.Y = 0
}
Agt_Step{
My.X = My.X + 1
}
```

policeは

```
Agt_Init{
My.X = 0
My.Y = 0
}
Agt_Step{
Dim set As Agtset
Dim one As Agt
My.X = My.X + 2
MakeAgtset(set, Universe.alley.thief)
one = GetAgt(set, 0)
If My.X >= one.X Then
    ExitSimulationMsgLn(GetCountStep() & " Steps")
End if
}
```

として下さい。policeがthiefに追いついた時点でシミュレーションが終わり、終了ステップ数が出力されます。

chaseと名付けて保存し、何度か実行してみて下さい。

ルールは全く同じなのに、追いつくまでに必要なステップ数がなぜか一定しません。これは、ステップ内の実行順序が「police → thief」になるか「thief → police」になるかがランダムに決まっているから起きる現象です。前項で見た「実行順序指定」を用いることで解決することもできるのですが、そうすると、thiefだけ全て実行し、policeだけ全て実行する、という順番になってしまいます。それが望ましくないときや、おなじエージェント種を追いかけるモデル（鬼ごっこなど）では「初回のみ実行順序を並び替え」オプションを使うと良いでしょう。ただし、「初回のみ実行順序を並び替え」としても、実行中にCreateAgt()でエージェントが新しく作られたり、DelAgt()やKillAgt()などで数が減ったりすると、もう一度シャッフルし直しますから注意して下さい。

- <失敗例C>有利不利の固定：同じ実行順序で繰り返すことによって生じる問題もあります。第16章の生態系のモデルを思い出して下さい。二つの動物プランクトンがひとつのエサを狙っていたとします。先に実行される方は、先に食べてしまう事ができるので有利です。同じエージェントを多数作ったつもりでいても、エサをとりやすいエージェントと、しわ寄せを食ってエサをとりにくくなるエージェントを意図せずに作り出します。こうなると、捕食機会が平等ならば全員が生き残れる量のエサがあっても、一部のエージェントが死んでしまうこともあります。この有利不利がモデル作成者の意図していない差別化で、それが結果に影響を及ぼしていたとしたら、モデルの作成上は失敗と言えるでしょう。

特に厳密な結果が求められる研究や実用では、実行順序は非常に重要なチェックポイントです。「こんなモデルにはこの順序」という処方箋を出したいところですが、モデルで表現したい事は千差万別ですから、一概に言う事はできません。ここでは代表的な失敗例をあげましたが、他にも意外な盲点があるかもしれません。モデルを作るときには、選んだ実行順序が自分の表現したいことにあっているかどうか、一度フローチャートを書きながら確認してみるのがよいでしょう。

31.3 初めと終わりの特別なステップ

GetCountStep()を使ってみるとわかるのですが、ステップは「0, 1, 2, ...n」と数えられています。上で説明したのは、1, 2, といったふつうのステップ内部の話です。これに対して、ステップ0は特別なステップで、初期化のためのルールであるUniv_Init{ }とAgt_Init{ }だけが実行されます。ツリーで数を3と指定してエージェントを作ると、ステップ0は、Univ_Init{ }をはじめに実行しそのあと3つのエージェントのAgt_Init{ }を実行します。このAgt_Init{ }の順番は、前節で説明したオプションで指定した順番に準じます。

ただし、ルール中にCreateAgt()で新たにエージェントを作ったときには、新しく作り出されたエージェントのAgt_Init{ }はCreateAgt()と書いた行に割り込んで、即時実行されます。したがって、Univ_Init{ }の中のCreateAgt()で生成したエージェントのAgt_Init{ }の方が、ツリーで作られるエージェントのAgt_Init{ }よりも早く実行されることになります。

次に、シミュレーションの最後のステップについて説明しましょう。シミュレーションを終える方法は3種類に大別できます。まずは手動で停止する方法です。停止ボタン、またはキーボードのEscキーを押します。すると、ふつうのステップ同様、Univ_Step_Begin{ }, Agt_Step{ }, Univ_Step_End{ }と実行されて、最後にUniv_Finish{ }が実行されて終わります。停止させた時点で即座に実行が中止されるのではなく、そのステップの最後までひと通り実行される事に注意して下さい。Univ_Finish{ }は、シミュレーションの試行がひと通り終わった後に、その記録を集計したり書き出したりするのによく使われます。必要がなければ、空白のまま残しておきましょう。

第二の方法は、ルール中で組み込み関数によって終了させることです。既におなじみのExitSimulation(), ExitSimulationMsg(), ExitSimulationMsgLn()のことです。これらの関数が実行されると、たとえそれがステップの途中であっても即座にUniv_Finish{ }に飛び、実行が終了します。エージェントのルールエディタで書いてあれば、それ以降のエージェントのルールは実行されません。ステップを一通り実行してから終了したい場合には、Univ_Step_End{ }にこれらの関数を書くようにルールを構成しましょう。

シミュレーションを終える第三の方法は、「実行環境設定」で終了条件を設定して、シミュレーションを

自動的に終える方法です。ステップ数、実行時間、もしくは条件式を書き込んでおけば、それが満たされたときに自動的に終了してくれます（詳しくは第35章を参照して下さい）。たとえばpersonの数が0になったら終了したいときには、「CountAgt(Universe.street.person) == 0」と条件式の部分に書き込めば良いだけです。この場合もステップの最後のUniv_Step_End{ }まで一通り実行されたあとに、Univ_Finish{ }に飛びます。

31.4 ステップをまたいだ実行順序制御のテクニック

第30章で作ったモデルでは、時間の流れが対戦、試合、世代という3つの階層から構成されていました。artisocでは時間はステップが刻んでいますから、それをもとにモデル作成者が重層的な時間を管理する必要がありました。

このように、複雑なモデルになると、一通りの動作を1ステップだけで書ききれないこともあります。その場合は、ステップをうまく利用して、時間の流れを管理し、ルールの実行順序を管理しなくてはなりません。たとえば、大学の入学試験のモデルを想像してみてください。以下のスケジュールを毎年繰り返します。

大学	受験生
(1) 募集を出す。	
(2)	希望分野と成績を考慮して、良いところから順にいくつか出願する。
(3) 入学試験をする。	受験する。
(4) 試験の優秀な人から一定数を合格、残りを不合格として通知を出す。	
(5)	複数の大学に合格したら、第2志望以降を辞退する。全て落ちたら浪人する。

試験をする前に合格通知を出したりすると大変な不祥事になってしまいますから、①②③④⑤の順番はきちんと守らなければなりません。しかしどうでしょうか？一通り動作するためには大学や受験生は相

手の反応を見ながら何度も行動しなくてはならないので、この章の冒頭で示した実行順序の設定を使っても、1ステップ内で①から⑤までを全て処理するのは困難です。そこで、1つめのステップで①を、2つ目のステップで②を……といった風に実行していき、5ステップで一通りの動作を終えたら1ターンとして、それを繰り返す必要があります。こんなときにはUniverseで5つの数字を繰り返し数えて、エージェントはそれにあわせて行動させましょう。まずは、直下にフェーズを表す変数を作り、その変数が
1→2→3→4→0→1→2→3→……というふうに変化するルールを書きます。

```
Univ Step Begin{
    Universe.phase = Universe.phase + 1
    If Universe.phase == 5 Then
        Universe.phase = 0
    End if
}
```

そして、エージェントルールでは、

大学	受験生
If Universe.phase == 1 Then ①のルール	If Universe.phase == 1 Then
Elseif Universe.phase == 2 Then	Elseif Universe.phase == 2 Then ②のルール
Elseif Universe.phase == 3 Then ③のルール	Elseif Universe.phase == 3 Then ③のルール
Elseif Universe.phase == 4 Then ④のルール	Elseif Universe.phase == 4 Then
Elseif Universe.phase == 0 Then	Elseif Universe.phase == 0 Then ⑤のルール
End if	End if

と書けばよいのです。ちなみに、大学のUniverse.phase == 2, 0および受験生のUniverse.phase == 1, 4では何もしませんから空白にしておくのがいいでしょう。これで、5ステップごとに①②③④⑤が繰り返し実行されます。

ちなみに、もっと単純な記述方法もあります。Modはその数を割った余りを示します。ですから、「GetCountStep() Mod 5」と書くと、ステップ数に応じて1→2→3→4→0→1→2→3→……と繰り返す数を得られます。

```
GetCountStep()           1,2,3,4,5,6,7,8,9,10···  
GetCountStep() Mod 5 1,2,3,4,0,1,2,3,4, 0···
```

これを利用すれば、Universeをいっさい使わずに、エージェントのルールの部分に

```
If GetCountStep() Mod 5 == 1 Then  
    ①のルール  
Elseif GetCountStep() Mod 5 == 2 Then  
    ②のルール  
    ·  
    ·  
    ·
```

と書き込むだけで、同じことを表現できます。もし「エージェントAは偶数ステップに行動する」としたい場合には、どこをどうすればよいか、もう分かりますね。

もっと変則的な実行順序も同じようにして簡単に記述できます。たとえば、選挙戦のモデルで、「90ステップかけて政治活動をして、最後の10ステップは選挙の期間。次の選挙まで90ステップはまた通常の政治活動をする」としたいときには、

```
If GetCountStep() Mod 100 <= 90 Then  
    通常ルール  
Else  
    選挙中のルール  
End if
```

と書きます。各フェーズのルールをFunctionやSub(第26章参照)にしてしまえば、さらにすっきりとわかりやすくなるでしょう。

(鈴木一敏)

新しく学んだ事項

- ステップ内の実行順序(ランダム、エージェント種別指定)
- 実行順序に関する典型的な失敗例
- 特別なステップ
- 複数ステップにまたがる疑似ステップを作る

第32章

モデルのミスをチェックする

32.0 モデルが何かおかしいときは

- ミスはつきものです
- 初歩的なミスはartisocが指摘してくれます
- artisocのデバッグ機能を活用しましょう
- 変数の中身をチェックします
- 一行ずつ実行して問題箇所をあぶり出します
- うまく動いているように見えても、油断は禁物です

32.1 ミスのいろいろなタイプ

前章まで、マルチエージェント・シミュレーションのモデルのさまざまな動かし方や作り方を学んできました。この章では、作ったモデルが、モデル作成者(つまり、私たち)の意図したとおりに動いているかどうかをチェックする方法を説明します。

実際に、新しいモデルを作成するときに、最初からモデル作成者の意図どおりのものがミス無くできることは、ほとんどありません。作ったものを動かしながら、ミスを見つけて修正しつつ、モデルの完成へと向かうことになります。ミスの見つけ方を学び、起こしやすいミスを把握することは、上手にモデルを作るうえで、とても重要なことです。

ミスには大きく分けて、

1. モデルが動かなくなるミス
2. モデルは動いているけれども、モデル作成者の意図とは異なる動きをしているミス

3. 一見、モデル作成者の意図どおりに動いているように見えるけれども、本当は異なる処理をしているミス

という3つのタイプがあります。

32.2 モデルが動かない

まず、第1のタイプのミスについて見てみましょう。このタイプのミスは、修正しなければ動かないため、最初に対処する必要があります。artisocには、そのままのルールではうまく動かない場合、その部分を自動的に指摘してくれる機能が備わっています。そのため、このタイプのミスへの対処は、比較的簡単です。

皆さんには、既にartisocのミスチェック機能に気づいていると思います。今までにまだこの機能の御世話になっていない人はいないでしょう。

モデルを実行しようとして実行ボタンを押したとき、第1のタイプのミスがあれば、artisocはシミュレーションを実行せず、その代わりにミスの存在を指摘してくれます。うまく処理できなかったルールの部分を、artisocがハイライトします。そしてその際に、ウィンドウの左下の欄外にartisocがうまく処理できなかった理由についてのメッセージが表示されます（[図32.1 参照](#)）。

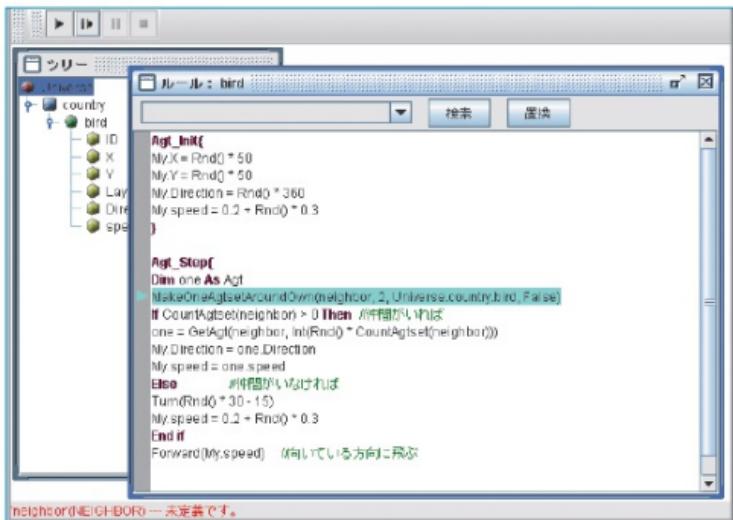


図32.1

これはミスを特定するのに参考になります。たとえば、「一未定義です」というのは宣言せずに変数を使っていたり、関数の綴りを間違えたりしてartisocが理解できなかった、ということです。また、しばしば、登場するのは、「(変数が)初期化されていません」という内容のメッセージです。ルールエディタの中で作成した「一時的な変数」は、変数の宣言をした時点では、初期値が定められていない状態(=初期化されていない状態)です。多くの場合、「変数=○○」という形で、ルールの中でその値が定められるため問題は起こりません。しかし、その値が定められる前に、「△△=関数(変数)」といったようにその変数を右辺で使用したり、イフ文の条件式で利用したりすると、このエラーが出ることがあります。そんなときは、変数宣言文の後、ルールの前の部分で、「変数=0」といったかたちで、変数の初期値を与えてやりましょう。

ここで注意すべきことは、ハイライトされる箇所は、ミスが原因となってartisocが処理できなくなった箇所であるということです。必ずしも、ミスの存在する場所ではありません。そのため、問題箇所が特定できないこともあります。こういう場合の最も基本的で効果的なやり方は、ルールの一部を凍結した状態でモデルを動かしてみることです。ある部分のルールがない(=凍結した)状態でモデルを動かしてみ

て、モデルが動けば、凍結した部分にミスを引き起こした何らかの原因があると考えられます。そして、その部分を詳細に検討することにより、ミスを発見することができます。

凍結する方法は非常に簡単です。凍結する部分の最初に「/*」最後の部分に「*/」と書き込んでやりましょう。その2つの記号に挟まれた部分は色が緑色に変わります。artisocは、この部分を、ルールではなくて、モデル作成者の「コメント」として無視します。そのため、実質的には、その部分を凍結したことになるわけです。凍結しようとしているルールが少ない場合には、各行の頭に「//」を付けてやることによって、それぞれの行全体がコメントとして取り扱われます。コメントをつける方法は、第11章やコラム[「ルールを見やすく」](#)で学びましたね。この方法を応用するわけです。

最初はルールの大部分を凍結してみて、モデルがまがりなりにも動くことを確認し、徐々に凍結する範囲を狭くしていきましょう。いずれミスのある場所が分かることになります。ただし、「Agt_Init{　}」や「}　」など、もともとあるべき文字や記号をコメントにしてしまわないよう注意してください。必ずエラーになってしまいます。

こうして無事にモデルは動くようになります。しかし、モデルがモデル作成者(私たち)の意図していたとおりに動かないことも、多々あります。その場合の対処方法について、次節以降で紹介します。

32.3 思ったとおりに動かない

ここからは、モデルは一応動くけれども、モデル作成者の意図(想定した動き方)とは異なる動きをするという第2のタイプのミスへの対処法について学びます。

出力範囲や出力要素を正しく選択しこねているために、本当はちゃんと動いているのに、そう見えていなかっただけという凡ミスから、ルールが処理されていく順番(実行順序、第31章を参照)の問題で、エージェントが周囲を正しく認識していなかったり、計測するタイミングの問題で、Universeの集計した値が本当の最終結果ではなかったり、といったモデルの設計に関わるミスまで、このタイプのミスは、本当に頻繁に起ります。

このタイプのミスは、ルールには文法上の誤りはなく、モデル自体は問題なく動きます。そのため、逆にミ

スの存在する場所を見つけるのが難しく、修正は容易ではありません。artisocでは、この作業を補助するための、幾つかの機能が備わっています。また、ミスを見つけるための便利な方法もあります。いずれも、私たちがモデルを作成するときに用いている、実際的で効果的な方法です。

最初に紹介するお手軽な方法は、第23章(23.5)で学んだエージェントの情報表示を用いる方法です。artisocのエージェントは、マップ出力画面において、それぞれが持っている変数の値を表示することができます。モデルが動くに従って、それぞれのエージェントの変数がどのような値をとるのかを追うことで、本当に意図したとおりに動いているのか、あるいは動いていないのはモデルのどの部分なのかを確かめることができます。**マップ出力設定** > **要素設定** で、「情報表示」にチェックを入れて下さい。そこで選んだ変数の値が、マップ出力のエージェントの上に表示されます。変数が意図されていないような異常な値をとっていないか、チェックしてみましょう。[図32.2](#)は、第26章で作成したboidモデルの鳥エージェントに、それぞれの速さの値を表示させたものです。

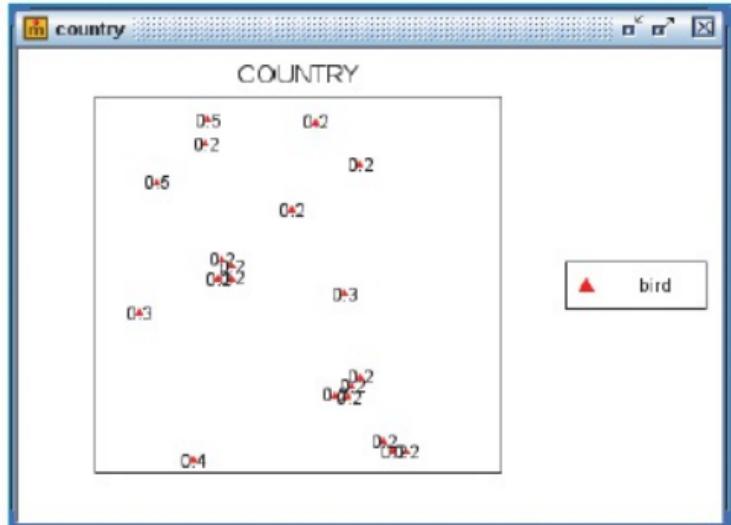


図32.2

変数の値を見るには、Print()関数を用いる方法もあります。こちらの方法のほうが、より厳密にモデルの動きをチェックすることができます。Universeやエージェントのルールの任意の場所でPrint()やPrintLn()を用いることで（Print()はPrintLn()と同じ機能で、改行されないもの）、その時点での特定の変数の値をコンソール画面に表示させることができます。「&」を用いて、複数の変数の値を同時に表示することも可能ですし、また「'''」を用いてメッセージと一緒に表示させることもできます（第8章8.5参照）。モデルが動いている途中で、気になる変数がどのような値をとっているのかを逐一、表示させることができるわけです。

ここで、ひとつ気をつけておかなければならないことがあります。マルチエージェント・シミュレーションのモデルには、通常同じエージェント種のエージェントが多数含まれています。あるエージェント種に、ある変数を表示するようなルールを書くと、そのエージェント種に含まれる全てのエージェントがそのルールを実行してしまいます。そのため、非常に多くの値が、一気にコンソール画面に表示されることになります。これでは、個別のエージェントがどんな動きをしているのか追跡するのは困難です。そこで、以下のようにして、特定のIDを持っているエージェントのみに、自分の変数を表示させる方法があります。

たとえば、第26章(26.5)のボイドモデルでAgt_Step{}のルールの最後に、

```
If My.ID == 0 Then
    Print(GetCountStep() & " steps: ")
    PrintLn("num_neighbor= " & CountAgtset(neighbor))
End if
```

と書けば、IDが0のエージェント（だけ）について、現在のステップとそのときに自分が認識した仲間の数をコンソール画面に書き出します。さらに、上で紹介した「情報表示」の機能を用いて、自分のIDをマップ上に表示させておけば、指定したIDのエージェントがマップ上でどういうふるまいをしているのかを、同時に観察することができ、モデルのふるまいを非常に効率的に検討することができます。[図32.3](#)のマップ出力の上では、鳥エージェントのIDが表示され、コンソール画面ではIDが0の鳥エージェントが各ステップに認識した周囲の鳥の数が表示されています。

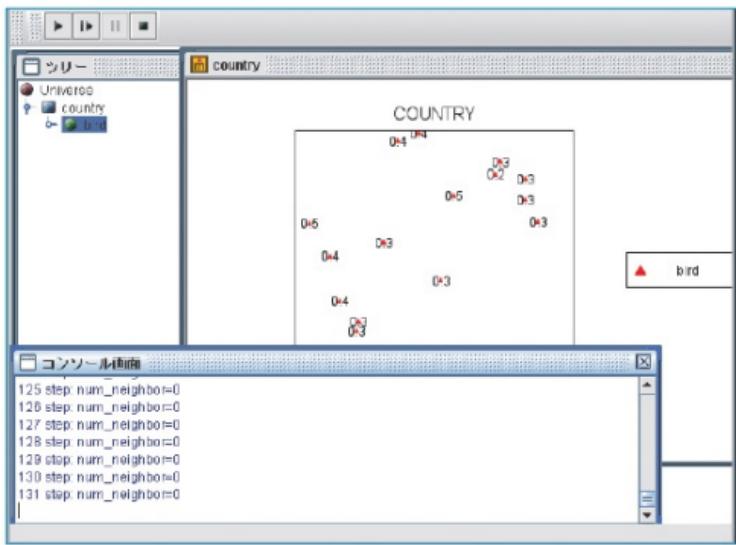


図32.3

32.4 変数の値をもっと詳しくチェックする

ここまででは、モデルを通常どおり動かしながら、変数のとっている値を表示させ、モデルがうまく動いているかどうかチェックする方法を学びました。この方法は、変数の値を出力させるポイントを設定する、いわば定点観測です。ルール内部の計算が正しく行なわれているか、正しく条件分岐しているかなどを事細かにチェックしたいときには、観測ポイントを増やさなければなりませんし、たとえ増やしても、観測ポイントの間でなにが起きているのかは結局わかりません。そこで、デバッガ機能という大変に便利なartisocの機能を紹介しましょう。

デバッガ機能とは、ルールを一行毎に実行させながら、それぞれの時点でのモデルの動きを確認することを可能にする機能です。早速使ってみましょう。好きなモデルを立ち上げた状態で、**デバッグ** のメニューから、**デバッガの起動** を選択します。モデルの背景が黄色に変わり、モデルがデバッガモードになったことが示されます。そして、「コンソール画面」とともに「デバッガ画面」というウインドウが表示されま

す。さらに、通常の「実行・ステップ実行・一時停止・停止」に代わり、「ステップイン・ステップオーバー・一時停止・続けて実行・停止(デバッガの終了)」のボタンが機能します。ステップインでは、ユーザー定義関数やCreateAgt()を用いたときに実行されるAgt_Init{}など、一見1つのルールのように見えて、実は別の箇所で実際の処理を行なっている場合に、その中に入り込んで全て1行1行実行します。一方、ステップオーバーは、関数は関数として実行して、中にまでは入り込みません。より詳細に見たいときはステップイン、細かなところをとばしてチェックする場合はステップオーバーを用いましょう。

では「ステップイン(ステップオーバー)」を押してみましょう。ルールエディタが表示され、ある行に青い矢印(マーカー)があり、その行全体が青くハイライトされます。これが現在、処理されているルールです。そのまま「ステップイン(ステップオーバー)」を押していくと、マーカーと青いハイライトが順次下に降りていくのが分かると思います。こうして、ルールが一行ずつ処理されていくのがわかります。繰り返し文などの部分では、同じ箇所を何度も処理しているのがわかると思います(図32.4参照)。

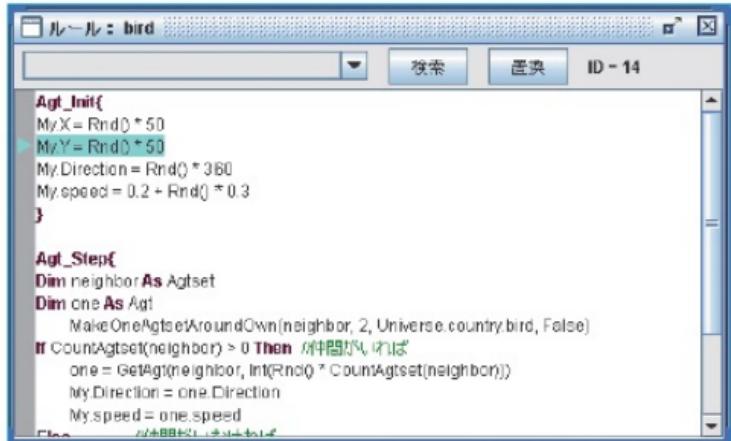


図32.4

デバッグ > デバッガの終了 を選択することで、デバッガモードを終了させて通常のモードに戻すことができます。

このようにartisocでは、モデルのルールが処理されていく様子を1行ずつコマ送りで見ることができますが、デバッグ作業において威力を発揮するのは「デバッグ画面」です。このデバッグ画面に変数名を入力し、エンターキーを押すと、その時点での変数が持っている値をコンソール画面に表示させることができます。試しに、青い矢印(マーカー)がエージェントのルールエディタにあるときに、デバッグ画面に「My.X」と入力し、エンターを押してみて下さい。その時点でのエージェントのX座標が表示されたはずです。この機能を使うことにより、1行毎にモデルのルールを処理させながら、それぞれの時点で正しく計算が行なわれているかをチェックできるのです。

デバッグ画面には、もうひとつ別の使い方があります。それぞれの時点で任意の変数に任意の値を格納させることができます。たとえば、空間の左端にさしかかった時のエージェントの挙動を調べたいときは、Agt_Step{ }の一番はじめにマーカーがあるときに、デバッグ画面に「My.X = 0」のように代入文を書いてエンターを押します。すると、そのエージェントのX値が0になります。そのままステップインを繰り返せば、左端にいるときの行動を観察できるのです。

このように、特定の状況においてエージェントがモデル作成者の意図したとおりに行動するかどうかを確かめることができます([図32.5](#)参照)。

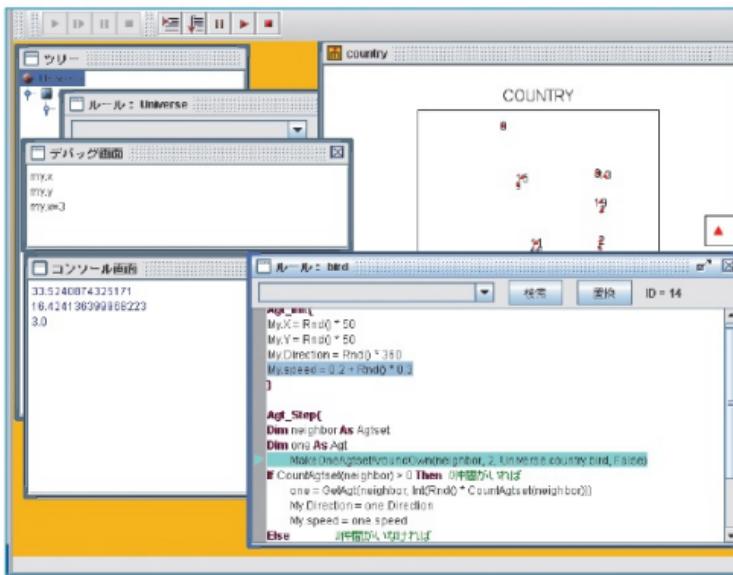


図32.5

32.5 より効率的に変数の値をチェックする

デバッガ機能は、モデルが意図したとおりに動いているかどうか、そして問題があるとすれば、どの部分に問題があるのかを細かくチェックすることのできる便利な機能です。しかし、実際にデバッガ機能を使ってみればわかりますが、モデルを1行1行動かしていくのは、非常に時間がかかります。とくに、多くのエージェントが含まれていたり、ルールの中に繰り返しが多かったりすると非常に長い時間がかかり、超人的な忍耐力と人差し指の持久力を必要とします（実際には耐えきれません）。

そのため、artisocのデバッガ機能には、「ブレークポイント」という機能があります。ルールの中で特にチェックしたい部分のみをゆっくり実行し、それ以外の部分はとばして実行させることができるのです。

チェックしたい箇所にカーソルを合わせて、選択してみましょう。その行が灰色にハイライトされると思いま

す。そしておいて、デバッグ > ブレークポイントの設定／解除 を選択すれば、そこにブレークポイントが設定されます。ルールの左に黄色い●が表示され、そこにブレークポイントが設定されていることを示します。解除する際には、もう一度その行を選択して デバッグ > ブレークポイントの設定／解除 を行なえば、そのブレークポイントは解除されます。デバッグ > 全てのブレークポイントの解除 を選択すれば、設定されている全てのブレークポイントが解除されることになります(図32.6参照)。



図32.6

ブレークポイントを設定しておいて 続けて実行 を選択する(または、実行ボタンを押す)と、モデルは通常どおりに実行され、ブレークポイントで止まります。そこで、「ステップイン(ステップオーバー)」によりブレークポイントの行を処理したあと、再び 続けて実行 を選択すると、ルールは通常どおりに処理され、次のブレークポイントで止まります。

ブレークポイント毎にモデルは止まるので、その時点での任意の変数の値などを、「デバッグ画面」と「コンソール画面」を用いることでチェックできます。

それでも、エージェントの数が多いときには、エージェントごとにモデルが止まってしまい、なかなかスムーズにモデルの動きをチェックすることはできません。そこで、条件設定付きのブレークポイントを設定するこ

とで、ある特定のエージェントの動きのみを追うことが可能となります。たとえば、

```
If My.ID == 3 Then  
    PrintLn("BreakPoint") //モデルに影響を与えない適当なルール  
End if
```

などとルールに追加した上で、PrintLn()の行にブレークポイントを設定します。そうすれば、IDが3のエージェントの時だけブレークすることができます。ほかにもたとえば、50×50の空間において、イフ文の条件式に

```
If My.X > 48 And My.Y > 48 Then
```

と書き込んで、右上の隅にいるエージェントの挙動だけを確認するなど、この方法はさまざまに応用できます。

32.6 思ったとおりに動いている？

これまで、第2のタイプのミス、つまりモデルは一応動くけれども、モデル作成者の意図とは異なる動きをする場合の対処方法について学びました。結局の所、モデルの挙動をモニターして、おかしな事になっている原因を絞り込んでいくしか方法はありません。人間にミスはつきものですから、大抵の場合ここまで紹介してきたような作業を経て、モデルはようやく作成者が意図したとおりに動くようになるのです。しかし、事前に予想していたとおりに動いていれば、それでよいのでしょうか？

人工社会というアプローチでは、モデル作成者は、局所的な相互作用については明確なイメージを持っていますが、全体のふるまいについてはぼんやりとしたイメージしか持たないことがしばしばです（それがこの技法の特徴です）。その相互作用の結果、思いもよらない面白いふるまいが引き起こされる場合があります。分居モデルやライフゲームなどの面白さはそこにあるのです。つまり、ミスがなくても予想外の動きをすることがあります。

逆に、予想どおりの動きをしていても実はそれがミスによって引き起こされている事もあるのです。これが

第3のタイプのミスです。つまり、意図したとおりに動いているように見えるときでさえ、モデルの内部で想定した処理が正しく行なわれているかどうかを逐一確かめなければなりません。ミスが存在するかどうかでさえ明らかでない状態で、正しいかどうかをチェックするのは非常に根気の要る作業です。しかし、研究や実用目的でモデルを作成する場合には、完全に想定したとおりに動いているという確信を持つ必要があります。本格的なモデルを作る上で、どうしても通らなければならない道です。

このタイプの典型的な失敗としては、実数を格納したいのに整数型変数を用いてしまう、「>」とすべきところを「>=」と書いてしまう、セル型の空間を想定しているのにMakeOneAgtsetAroundOwnCell()のCellを付け忘れてしまう、条件文のAndとすべき所でOrと書いてしまう、などがあります。

また、特に注意しなければならないのは、実行順序です。実行順序はルール上に文字として表れませんから忘れてしまいがちですが、前章で紹介したように、実行順序に注意を払わないと生じる失敗もあります。こうした間違いでは、正しい場合と似たように動いてしまう事も多々あります。一見正しく動いているように見えるモデルでもチェックする必要があるのです。

第3のタイプのミスを見つけるためには、これまで学んだ方法を組み合わせて、丹念にチェックするしか方法はありません。特にモデルを作成している途中にチェックを繰り返すことは、非常に有効な方法です。一気に全部のルールを書き上げるのではなく、一部のルールを書き上げるたびにモデルを試しに走らせ、Print()やPrintLn()を用いてコンソール画面に変数の値を表示させたり、あるいは、デバッガを使ってチェックしたりします。ルールを書くときに、ユーザ定義関数を使ったりしてルールをすっきりさせておくと、このようなチェックを行ないやすくなります。ルール内の処理を書き上げるごとに、その部分についてミスのチェックを行なっていけば、完成度の高いモデルを作り上げることが可能です。面倒くさいように思えるかもしれませんが、結果的には、このほうが時間の節約になることが多いと思います。経験的にも、どんな小さなモデルであっても、途中でチェックを繰り返しながらモデルを作成したほうが、結果的には早くできあがることが多いようです。ミスが些細なものであればあるほど、あとで長いルールの中から探し出すのは大変なものなのです。

(光辻克馬・鈴木一敏)

新しく学んだ事項

- artisocのルールチェック機能
- コメントアウトで問題箇所を絞り込む
- マップ上の情報表示をミス探しに利用する
- Print(), PrintLn()で変数の中身をモニターする
- デバッグでルールを1行ずつチェックする
- デバッグ画面で変数の中身を覗く、見える
- ブレークポイントを設定して効率を高める
- 動いたからといって安心しない

実数と擬似実数

「コンピュータが実数を扱えない」といわれると奇妙に聞こえるかもしれません。自然数(そんなに大きくな)や0.1のようなキリの良い数だったら、計算できます。しかし、円周率のように無限に続く数値は、有限の世界にあるコンピュータ上に正確に表すことができません。スーパーコンピュータを何週間も使って1兆桁以上計算しても割り切れないのですから。そこで、コンピュータは桁数の多い実数(無理数だけでなく有理数でも)を扱うとき、それに近い値で代用して計算しています。私たち人間は、半径20mのミステリーサークルの面積は $400\pi m^2$ と「計算」できますが、これだって厳密に考えたら数値の計算を先送りしているだけにすぎません。どのくらいの広さか考えるときには、コンピュータと同じように大体の値として3.14をかけてみてはじめて、「1m²のマスを1256個か。そんなにつぶすなんて大変だ」と実感を持って真夜中の重労働を想像できるわけです。

コンピュータのなかでは(artisocに限らず)、実数型の変数と称しても実は近似値(擬似実数)を用いて数値を表していますから、計算によって誤差が出ることがあります。誤差は非常に小さく、多くの場合シミュレーションに影響がないでしょう。しかし、「==」を使ってぴったり同じかどうかを判定するときには1ちょうど1.000000000000001は別の値と判断されてしまいますから注意が必要です。

同じ事は、実数型の変数を使う組み込み関数にも言えます。たとえば、aが自分から2離れているbに向かってforward(2.0)として進んだとしても、誤差のためa.X == b.X, a.Y == b.Yにならないことがあります。こんなときは、a.X = CDbl(Round(a.X*1000))/1000のように書きましょう。Round()は小数点以下を四捨五入する関数ですから、上の場合、小数第四位以下を四捨五入したことになり、それ以下の誤差を無視することができます。

(鈴)

第33章

画像をマップに表示する

33.0 見た目も大切です

- 背景やエージェントを画像で表示できます
- 画像は状況に応じて切り替えられます
- エージェントの初期値データをファイルから流し込めます

33.1 見栄えも理解を助けます

これまでに学んだ技法を用いれば、かなり本格的なモデルを作成することができるはずです。内容が本格的になれば、見た目にもこだわりたくなるのが人情でしょう。とはいえ、見た目にこだわる事は、単に格好付けのためだけではありません。第23章の文化変容モデルで色が使えなかったらどうなるのか、想像してみてください。シミュレーション結果をわかりやすい形で示す事の重要性がわかってもらえたと思います。

この章では、背景やエージェントに色を付けるという所から一步進んで、背景やエージェントを画像で示す方法を紹介します。これらの技法は、実務で用いられるような本格的なモデルでも利用されています。背景画像については、交通モデルの道路地図を背景として表示する、避難誘導のモデルで、洪水によって現実の市街地が徐々に冠水する様子をステップ数に連動させて映し出す、などの例があります。また、エージェントを画像で表すことで、向きや大きさなど、●や■のアイコンだけでは表しきれなかった属性を一目瞭然に示すことができます。さらに、エージェントの座標を背景画像と関連付ける方法として、エージェントの変数の初期値をファイルから読み込む方法も紹介します。

この章ではサンプルのファイルを使います。まずはその準備として、山影研究室webサイトにあるartisoc教科書ページ(<http://citrus.c.u-tokyo.ac.jp/artisoc/textbook>)から33materials.zipをダウンロードして、デスクトップに解凍しておいて下さい。33materialsというフォルダが展開されるはずです。

それでは、新しいモデルを作りながら、二次元マップの背景に画像を用いる方法を紹介します。

背景に地図を表示する

1. Universeの下に200×100の横長の空間、globeを作ってください。他の設定はデフォルトのままでかまいません。
2. globeをマップ出力させる作業をします。その際、マップ出力設定の「背景画像」の右側のチェックボックスをクリックしてみて下さい。ファイル名を書き込む欄が出てきたと思います。そこにback.gifと書き込み、**了解**を押して下さい。
3. モデルはmapという名前を付けて保存します。
4. つぎに、33materialsの中にback.gifというファイルが入っていますから、これをmap.modelのあるフォルダにコピーして下さい。

これで終了です。実行ボタンを押せば、マップの背景に画像が表示されます。

練習として、自分の好きな画像を表示してみましょう。artisocはjpg, gif, pngの三種類の画像フォーマットに対応しています。背景に用いたい画像がこの三種類以外の場合には、画像編集ソフト(Windowsならばペイントなど)を使って変換して、モデルファイルと同じフォルダに入れて下さい。画像ファイルの名前を、出力設定の背景画像の欄に書き込みましょう([図33.1](#)参照)。マップの大きさはウィンドウの大きさによって変わりますが、背景画像はマップの大きさに自動的にフィットしますから大きさは自由です。ただし、正方形のマップに横長の画像を使うと、横幅が圧縮されてしまいます。mapモデルの空間のサイズをプロパティを変更していろいろな大きさ(たとえば100×100)にしてみるとよくわかります。画像をそのままの形で表示したければ、マップの縦横比と画像の縦横比を合わせるようにしましょう。



図33.1

33.3 エージェントを画像で表す

同様に、エージェントも■や●の代わりに画像で表す事ができます。この場合も、jpg, gif, pngのいずれかの画像をモデルファイルと同じフォルダに入れておいて、マップ出力設定の出力リストの要素設定で、背景画像と同様にファイル名を入れます。

試しにモデルを作ってみましょう。画像を表示する技法を学ぶだけですから、モデルは簡単なものでよいでしょう。

マップ上を零戦が真っ直ぐ飛ぶ。

1. 新規フォルダを作り、33materialsフォルダの中身のzero.gifという画像ファイルをコピーしておきます。
2. Universeの下にskyという空間（設定はデフォルト）を作り、その下にzeroというエージェントを1体作って下さい。
3. 出力設定を行ないます。要素設定のマーカーのところで「ファイル名」を選択し、「zero.gif」と書き込んで下さい（[図33.2](#)参照）。



図33.2

zeroのルールには、

```
Agt_Init{  
MoveToCenter()  
}  
Agt_Step{  
Forward(1)  
}
```

と書き込んで下さい。実行環境設定で実行ウェイトを100ミリ秒ほどに設定するとよいでしょう。

画像と同じフォルダに、モデルをzerofighterという名で保存して、実行してみましょう。

ところで、エージェントの画像の場合、背景とちがって画像の大きさがそのまま表示されます。たとえば、32×32ドットの画像を使えば、16×16の画像を使ったときと比べて、縦横が2倍の大きさで表示されます。このため、車と人のように、大きさの違うエージェントを違和感なく表すことができます。一方、マップの大きさはウインドウの大きさ次第ですから、アイコンがマップに対して大きすぎる場合も出てきます。ウインドウを広げるなどして調整してみて下さい。しかし、エージェントにあまりに大きな画像を使ってしまうと、調整

しきれない事もあります。そんなときは画像編集ソフトを用いて、画像自体を適度に縮小してから使いましょう。

また、sky.jpgという画像をコピーして、前節と同じやり方で背景画像も表示してみましょう。

33.4 状況に応じて画像を切り替える

背景やエージェント画像はルール内で切り替えることができます。まずは、背景画像を切り替える方法を紹介します。

ファイル名を入れるための文字列型の変数をあらかじめUniverseに作っておきます。それでは、上のzerofighterモデルのUniverseにbgという文字列型変数を追加して下さい。マップ出力設定の「背景画像」のボックスをチェックしてから「ファイル指定変数名」のボックスをチェックします。すると選択可能な変数(つまり文字列型)が選べるので(この例ではbgしかないはずです)、その変数を選びます([図33.3](#))。次に、33materialsフォルダから、day.pngおよびnight.pngの二つをモデルファイルと同じフォルダにコピーして下さい。そして、

```
Univ_Step_Begin{
If GetCountStep() Mod 1000 <= 500 Then
    Universe.bg = "day.png"
Else
    Universe.bg = "night.png"
End if
}
```

と書き込んで、bgの中に画像のファイル名を入れれば、500ステップ毎に昼と夜の背景画像が切り替わります([図33.3](#)参照)。



図33.3

同様に、エージェントの画像も切り替える事ができます。マップ上でエージェントの属性を表すとき、今まででは色を付けてきましたが、方向など、色だけではわかりにくい属性もあります。エージェントに文字列型変数(下の例ではicon)を追加し、マップ出力設定の要素設定で、「ファイル指定変数名」をチェックして、その文字列型変数を指定します。そして、ルール中で、その変数の中に、1.jpgなどとファイル名を入れればよいのです(図33.4参照)。

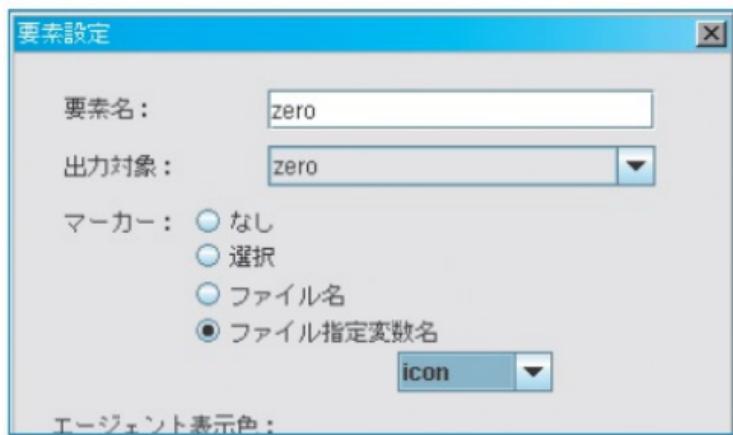


図33.4

ファイル名の入れ方ですが、まず、素直な方法から説明しましょう。上下左右を向いた四つの画像でエージェントを表したい場合、右に向いた画像を0.jpg、上向きを1.jpg、左向きを2.jpg、下向きを3.jpgといった名前にして、あらかじめモデルファイルと同じフォルダに入れておきます。そして、Agt_Step{ }の最後の部分に

```
If My.Direction < 45 Then  
    My.icon = 0.jpg  
Elseif My.Direction < 135 Then  
    My.icon = 1.jpg  
Elseif My.Direction < 225 Then  
    My.icon = 2.jpg  
Elseif My.Direction < 315 Then  
    My.icon = 3.jpg  
Else  
    My.icon = 0.jpg  
End if
```

と挿入すれば、自分の方向にあった画像を表示することができます。最後のElseの部分では、Directionが315~360までの値のときに右向きにするために、0.jpgを入れています([図33.5左図参照](#))。

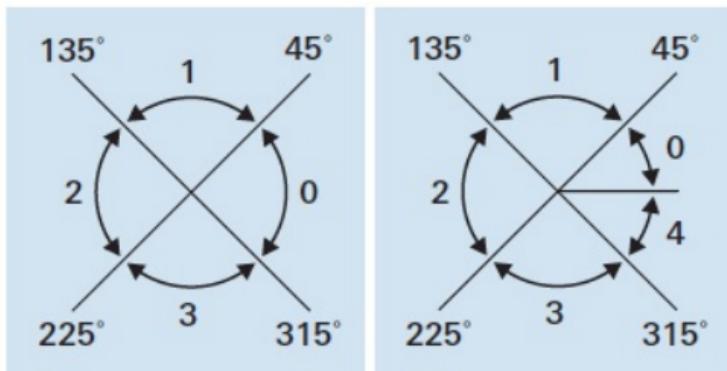


図33.5

このようにして非常に簡単に簡単な場合分けで表示画像を切り替える事ができるのですが、8方向、16方向と

分割数が増えていくと、ルールが非常に長くなってしまいます。そこで、もっと簡潔に書ける技巧的な方法も紹介しておきましょう。まず、もうひとつ右を向いた画像を4.jpgと名付けてフォルダに入れておきます。そして、

```
My.icon = CStr(Int(1 + (My.Direction - 45) / 90)) & ".jpg"
```

とします。すると、右辺第一項は図33.5右図のよう、Directionによって0から4までの数字が計算されます。そしてそれに拡張子「.jpg」を足したものがMy.iconに代入されます。本当に図のような数字が出てくるか、Directionの値をいくつか代入して計算してみて下さい。このDirectionを整数に置き換える部分は、次のように、もう少し一般的にまとめる事ができます。

```
Int(1+ (My.Direction - (360 / 分割数の二倍)) / (360 / 分割数))
```

8分割なら、0～22.5度は0、22.5～67.5度は1、……337.5～360度は8になります。16分割でも32分割でも同じように書く事ができます。そのぶん画像を準備しなくてはならないのが難点と言えば難点ですが。

では、実際に先ほど作った零戦が旋回するようにモデルを修正して、どちらを向いているのかわかりやすくしてみましょう。

1. 画像ファイルzero0.gifからzero.8.gifまでを、モデルファイルと同じフォルダにコピーして下さい。
2. zeroにiconという文字列型変数を追加して下さい。この中に画像ファイルの名前を入れます。
3. マップ出力設定の要素設定で、「ファイル指定変数名」をチェックしてiconを指定します。

ルールは、

```
Agt_Init{
MoveToCenter()
My.Direction = Rnd() * 360
}
Agt_Step{
Turn(5)
Forward(0.1)
My.icon = "zero" & CStr(Int(1 + (My.Direction - 22.5) / 45)) &
".gif"
```

}

でよいでしょう。zerofighter(B)として保存して実行してみて下さい。どうでしょうか、それらしく動いたでしょうか? この方法は、地下道の中での人の流れや鳥や魚の群れなど、エージェントの方向がわかった方が良いモデルに応用が利くでしょう。また、Directionによって画像を切り替えるだけでなく、資産や力によってエージェント画像の大きさを変えるなど、いろいろな活用方法を考えてみて下さい。

33.5 初期値を読み込む

エージェントの初期配置を背景画像に合わせる方法も紹介しておきましょう。たとえば、世界地図上に主要な国の首都を表示してみましょう。冒頭で作ったmapモデルに、エージェントとして、capitalというエージェントを148個追加して、出力設定して下さい。このまま実行すると、左下の南極あたりに全てのエージェントが重なって出てきます。このエージェントに、初期値として、緯度と経度のデータを与えましょう。

主要国の首都の緯度経度のデータを、artisocに読み込める形にしたのが、33materialsに入っているlocation.csvというファイルです。テキストエディタで開くと、中身はこんな風になっています。

```
"ID","X","Y","Layer","Direction"  
0,111.0092593,72.94444444,0,0  
1,107.3611111,45.09259259,0,0  
2,65.66666667,59.62962963,0,0  
3,66.66666667,29.72222222,0,0  
4,124.7314815,72.31481481,0,0  
.  
.  
.
```

一行目は、変数名が引用符に囲まれ、コンマで区切られて並んでいます。二行目以下では、各データの間はコンマで、各エージェントのデータは改行で、それぞれ区切られています。

ツリーのcapitalを選択して 設定 から 初期値設定 を選択、または、右クリックして 初期値設定 を押すと、「エージェント初期値設定」の画面が出てきます(図33.6)。右下の ファイル入力 を押して location.csvを選択して下さい。初期値が読み込まれましたね。実行ボタンを押すと、主要国の首都の位置にエージェントが表示されたはずです。うまくいかない場合は、ツリー上の変数名とファイル内の変数名が大文字小文字まで完全に一致しているかどうか確認してみて下さい。

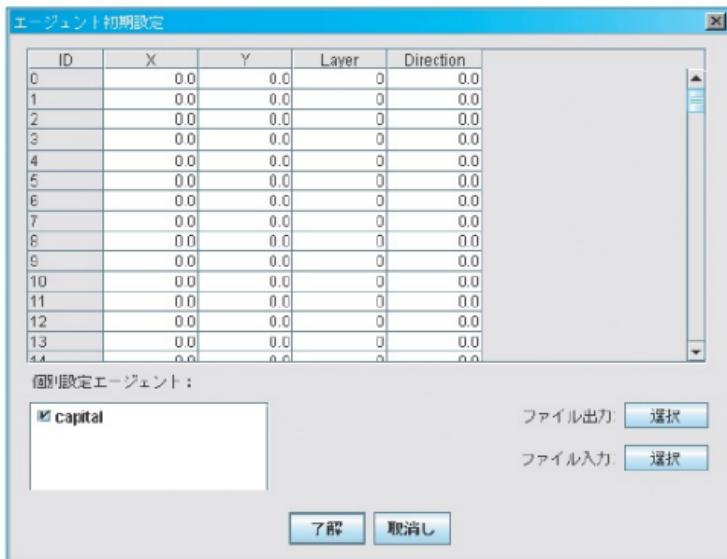


図33.6

自分で新たな読み込み用のファイルを作りたいときには、奇妙に思うかもしれません、まず ファイル出力 を押します。拡張子が.csvのファイルが新しく出来ますから、それをエクセルなどの表計算ソフトで開いて編集します。そして、上のような手順で読み込みます。こうすると便利です。

(鈴木一敏)

新しく学んだ事項

- 背景に画像を貼り付ける
- エージェントを画像で示す
- 画像をルールで切り替える
- エージェントの初期値をファイルから読み込む

第34章

空間変数を活用する

34.0 空間を利用するとモデルに深みが出ます

- 第18章で学んだ空間変数の応用です
- 空間変数の設定のみで雪を降らせましょう
- 船の航跡を描き、徐々に消える様子を表現しましょう
- 粒子をランダムに飛来させ、不思議なフラクタル図形を描いてみましょう
- 富士山への登山にも挑戦します

34.1 空間変数をもっと活用しましょう

第18章で、空間に空間変数というものを設定し、それをエージェントが利用したり、マップ出力で画面表示させたりする方法を学びました。そこでも触れたように、空間変数の利用の仕方には、いろいろなバリエティがあります。この章では、それらをまとめて紹介します。ステップ毎に空間の色が変化していくもの、多数のエージェントを出現させる代わりに空間変数を利用するモデル、あるいは空間変数の「勾配」によってエージェントの行動が変わってくるものなど、さまざまな空間変数の利用法を学習しましょう。

なお、この章のモデルでは、空間の大きさに左右されない一般的なルールになっています。つまり、空間の大きさを予め決めておくのではなく、ルールの中でその大きさを調べて、空間変数を操作する手法を用いています。

34.2 一面に雪を降らせる

空間に雪をランダムに降らせ、時間が経過するにつれて積もり、融けていく。

では早速モデル作成を開始しましょう。

1. Universeの下に空間100×100(他はデフォルト)のsnowlandを追加し、そこに整数型変数snowを追加します。
2. コントロールパネルで降雪量と融ける速さを設定できるようにするために、Universeの下に整数型変数であるaccumu(降雪量)とtemp(気温)を追加します。コントロールパネルのスライドバーでaccumuは0から10まで1刻みで、tempは0から5まで同じく1刻みで設定できるようにして下さい。accumuの値が大きいほど雪は積もりやすくなり、tempの値が大きいほど雪は融けやすくなります。
3. マップ出力を設定する。変数snowのマーカーは「なし」にして、変数範囲は0から10までとして、0は任意で、10を白色(RGBは255, 255, 255)に設定します。つまり10以上で雪が完全に積もって白色しか見えない状態、全くない状態が0となります。のりしろを消すためにX軸、Y軸ともに最大値を99としておきましょう。
4. モデルをsnow-countryという名前を付けて保存してください。

ルールエディタには、2分の1の確率でsnowlandに雪が降るように以下のように記述します。

```
Univ Step Begin{
Dim i As Integer
Dim j As Integer
//雪が降るとともに徐々に融けていく。
For i = 0 To GetWidthSpace(Universe.snowland) - 1
    For j = 0 To GetHeightSpace(Universe.snowland) - 1
        If Rnd() * 2 >= 1 Then //2分の1の確率で積もっていく
            Universe.snowland.snow(i, j, 0) =
Universe.snowland.snow(i, j, 0) + Universe.accumu
        End if
        Universe.snowland.snow(i, j, 0) =
Universe.snowland.snow(i, j, 0) - Universe.temp
        If Universe.snowland.snow(i, j, 0) <= 0 Then
            Universe.snowland.snow(i, j, 0) = 0
        End If
    Next j
Next i
}
```

コントロールパネルを動かして、いろいろな場合を試してみましょう。雪がしんしんと降っては積り、徐々に

融けていく様が再現できましたか？

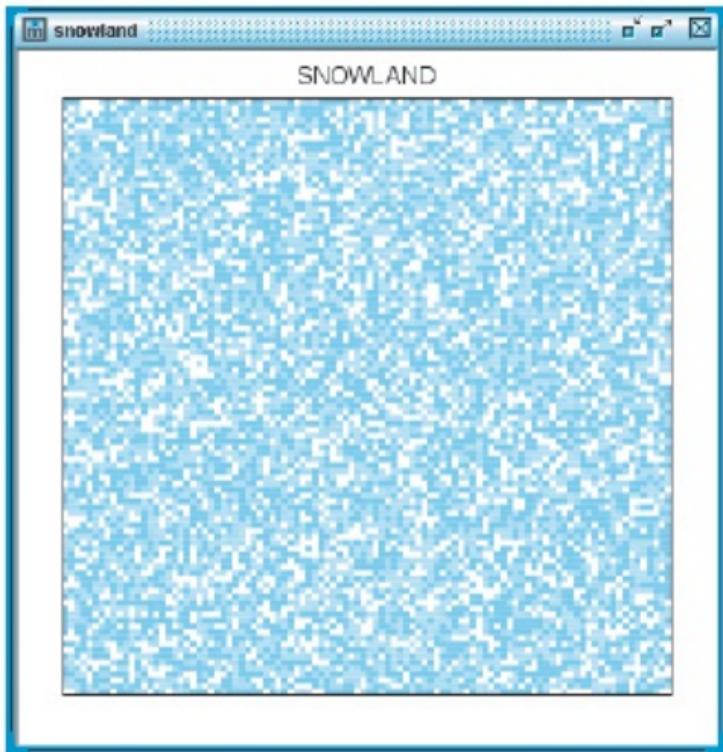


図34.1

34.3 航跡を描く

次は海上に船を航海させるモデルを作成しましょう。

船は海の中央からスタートして、毎ステップ0.5ずつ左右20度の範囲でランダムに動く。船が通った場所には航跡（英語でfurrow）が描かれる。

1. Universeの下に空間seaを追加(デフォルト)し、そこに整数型変数furrowとエージェントshipを追加(エージェント数は1)します。
2. マップ出力でshipとfurrowの両方を出力するように設定します。furrowのマーカーは「なし」にして、変数範囲を0から1に設定して下さい。0に通常の海の色を、1に航跡の色を指定しますが、たとえば0を濃い青色に、1を白色にすると、霧囲気ができます。のりしろを消すためにX軸、Y軸ともに最大値を49としておきましょう
3. モデル名をvoyageと名付けて保存して下さい。

エージェントshipのルールエディタには、以下のように記述します。最後のUniverse.sea.furrow(My.X, My.Y, 0) = 1が、空間に跡を残していくルールです。極めて単純ですね。

```
Agt_Init{  
MoveToCenter()  
My.Direction = Rnd() * 360  
}  
Agt_Step{  
Turn(Rnd() * 40 - 20)  
Forward(0.5)  
Universe.sea.furrow(My.X, My.Y, 0) = 1  
}
```

shipが見えにくい場合は、マップ要素リストの順序を変更して下さい(これは第18章(18.4)で学びました)。船が通った後に航跡が描かれたでしょうか?

ただしこのままだと航跡は消えず残ってしまい、不自然です。ある程度の時間が経てば航跡は徐々に消えるようにしましょう。つまり、空間変数の値が0(航跡がない)ならば値はそのままで、航跡があれば徐々に元に戻っていくというモデルに変更します。

マップ出力のfurrowの変数範囲を0から15に設定し、voyage(B)という新しい名前を付けて保存しましょう。

Universeのルールに以下のように記述します。前節の降雪モデルの応用です。

```

Univ_Step_Begin{
Dim i As Integer
Dim j As Integer
//航跡を少しづつ元の状態に戻す。
For i = 0 To GetWidthSpace(Universe.sea) - 1
  For j = 0 To GetHeightSpace(Universe.sea) - 1
    If Universe.sea.furrow(i, j, 0) <= 0 Then
      Universe.sea.furrow(i, j, 0) = 0
    Else
      Universe.sea.furrow(i, j, 0) = Universe.sea.furrow(i,
j, 0) - 1
    End if
  Next j
Next i
}

```

エージェントルールは以下のように変更します。

```

Agt_Step{
Turn(Rnd() * 40 - 20)
Forward(0.5)
Universe.sea.furrow(My.X, My.Y, 0) = 15
}

```

これで船が通った跡は徐々に消え、15ステップ後に完全になくなるというモデルになりました。

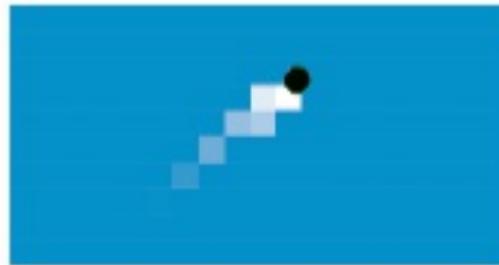


図34.2

34.4 DLA(拡散律速凝集)をつくる

1粒子が空間の中央に存在している。粒子が周囲から登場して、ランダムに動き回るが、既に空間に存在している粒子にぶつかったら、動き回っていた粒子はそこで停止する。これを繰り返す。

この非常に単純なルールで、フラクタル図形が形成されます。フラクタル図形とは、フランスの数学学者ブノワ・マンデルブロが名付けたもので、海岸線に代表されるような、部分的に見ても全体的に見ても複雑に入り組んだ形状をしており、それら部分と全体が自己相似形になっている図形のことを言います。

これから、モデル化の作業に入りますが、分子(粒子)をエージェントとして次々と生成するのではなく、停止した分子を新しく登場する分子に使い回しながら、空間変数を活用します。こうする理由は後で説明します。

1. Universeの下に空間roomを、たとえば 150×150 で(大きければ大きいほど曲面がスムーズなフラクタルができますが、その分時間がかかります)設定(他はデフォルト)して、エージェントmolecule(分子)を追加(エージェント数は20)して下さい。moleculeのXとYのプロパティを開いて、記憶数をそれぞれ1に変更します
2. 空間roomの下に整数型変数putを追加します。この変数によって、分子がその場所に止まっているのかどうかを区別させます
3. moleculeとputをマップ出力します。マーカーは双方とも「なし」にし、色も同じものにしましょう。putの変数範囲は0から1で、0は白色に、1の方をmoleculeと同色に設定しておきます。つまり、空間変数の値が1であれば、moleculeエージェントが存在するとみなします
4. 次にステップ数を、値出力画面で出すようにしましょう。出力項目リストの「値画面出力」を選択し、追加をクリックします。出力名、値画面名を入力してから、追加をクリックします。出力値にはGetCountStep()と記入します。これでステップ数が値出力画面で表示されるようになります。
5. このモデルをDLAと名付けて保存しましょう。

いよいよルールの記述に移ります

まず、Universeには、以下のルールを記入します。

```
Univ_Init{  
    Universe.room.put(GetWidthSpace(Universe.room) / 2,  
    GetHeightSpace(Universe.room) / 2, 0) = 1  
}
```

こここのルールは、まず空間の中央に分子を1つ置いたことになります。実際のmoleculeエージェントではなく、空間変数をその代わりにしていることに注意して下さい。

上書き保存して、実行してみましょう。空間roomの中央と左下に分子がいることを確認しましょう（正確には中央はmoleculeエージェントではなく、空間変数による「分子の存在」状態ですが）。また、ステップ数が毎回カウントされているでしょうか？

次に、エージェントルールに移ります。エージェントは、空間の縁辺のランダムな位置から出現し、ランダムな方向へ走ります。絶えず方向をランダムに変えながらセル型で動き、自分の進む方向に色が着いていたら(putの値が1ならば)他の分子とぶつかったとみなし、そこで止まります。

ここでは、第26章で学習した、ルールをまとめる手法を使用しています。空間の四隅からエージェントがランダムに現れるという初期ルールを、appear()というSubのタイプのユーザ定義関数で作成しています。ユーザ定義関数に値を受け渡す必要がないため、関数に続く括弧の中には何も変数が入っていません。

また、第19章で学習した「過去参照」法も使用しています。具体的には、エージェントはランダムに動き、自分の今いる座標に色が付いていれば、一期前の自分の位置に色を付けて、再び四隅から出現します。このようなルールにすることによって、他の分子にぶつかったら止まる、という行動を表現させています。

分子の樹木が空間の端にまで広がったら、シミュレーションを終了させるというルールもここで記入しています。

```
Agt_Init{  
    //出現するルール  
    appear()  
}
```

```

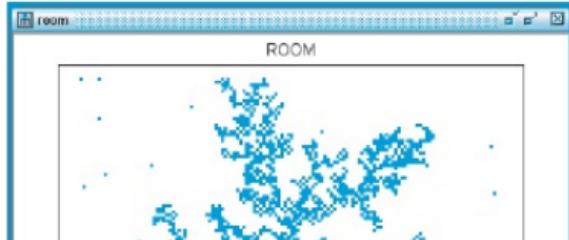
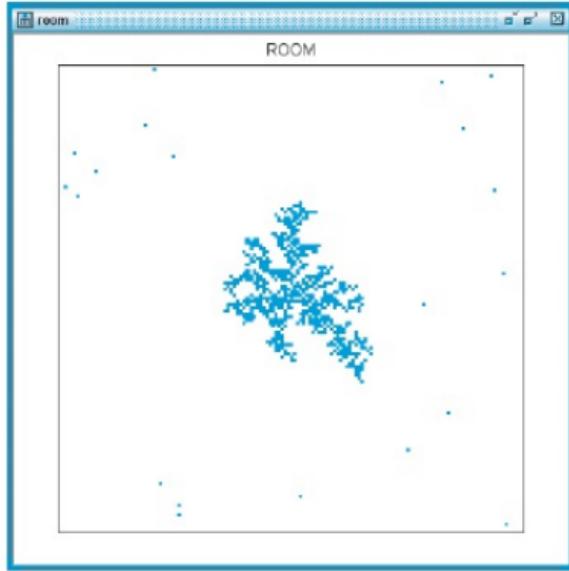
Agt Step{
//進む方向はセル型で、ランダムに
ForwardDirectionCell(Int(Rnd() * 8), 1)
If Universe.room.put(My.X, My.Y, 0) == 1 Then //自分の位置に色が付いていれば
    //色が縁辺に達していればシミュレーションを終了させる
    If My.X == 0 Or My.X == GetWidthSpace(Universe.room) - 1 Then
        ExitSimulationMsg("Completed after " & GetCountStep() & " steps")
    Elseif My.Y == 0 Or My.Y == GetHeightSpace(Universe.room) - 1 Then
        ExitSimulationMsg("Completed after " & GetCountStep() & " steps")
    End if
    //上記以外なら1ステップ前の位置に色を落とし、初期化する
    Universe.room.put(GetHistory(My.X, 1), GetHistory(My.Y, 1), 0)
= 1
    appear()
End if
}
//空間の縁辺にランダムに配置するユーザ定義関数
Sub appear(){
Dim i As Double
i = Rnd()
If i < 0.25 Then
    My.X = 0
    My.Y = Int(Rnd() * (GetHeightSpace(Universe.room) - 1))
Elseif i < 0.5 Then
    My.X = Int(Rnd() * (GetWidthSpace(Universe.room) - 1))
    My.Y = 0
Elseif i < 0.75 Then
    My.X = GetWidthSpace(Universe.room) - 1
    My.Y = Int(Rnd() * (GetHeightSpace(Universe.room) - 1))
Else
    My.X = Int(Rnd() * (GetWidthSpace(Universe.room) - 1))
    My.Y = GetHeightSpace(Universe.room) - 1
End if
}

```

ではDLA(B)と名付けて、実行させてみましょう。空間の四隅から現れたエージェントがのろのろと動き回ります。ただしこのままだと中央の分子にぶつかるには多大な時間を要するのは一目瞭然ですね。そこで、実行速度を速めた上で、毎ステップではなく、たとえば10000ステップ毎の結果を見るように設定を変更します。

設定 > 実行環境設定 を選んで、「実行ウェイト」を0ミリ秒にしましょう。これで実行の速度が速まります。また、出力タイミングを「ステップ毎」の10000と入力しましょう。この結果、さらに実行速度が速まります。DLA(C)と名付けて、保存しましょう。

いかがでしょうか？ 結果の表示は10000ステップ毎に現れるようになり、どんどん分子がくっついていき、フラクタルが形成される様子が観察できたでしょうか？ このパターンはいわゆる収穫遙増と同じで、長い分子の枝は短い枝よりも急速に成長していきます（[図34.3](#)参照）。



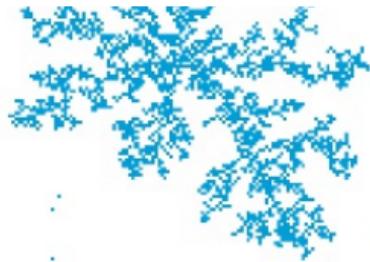


図34.3

このモデルの特徴は、moleculeエージェントが静止している他のエージェントとぶつかったら、そのまま静止させ、新しいエージェントを生成するのではなく、空間変数を用いて跡を記録し、エージェントは使い回しすることにあります（エージェント数は20で変わりません）。なぜこのようなルールにする必要があるのでしょうか？

空間 150×150 の中に、仮に10分の1のスペースにエージェントが存在していると想像してみましょう。エージェント数は2250に膨れ上がり、確実に実行速度は遅くなります。それを上のルールのように空間変数で代用することで、動きはほとんど変わらなくなるのです。この点が、空間変数を使用する大きなメリットです。

34.5 地形図のデータをマップに反映させる

この章の最後のモデルとして、富士登山モデルを完成させましょう。空間変数を活用した複雑なモデルですが、大変重要で汎用的なルールがあちこちで使われています。是非、挑戦してみましょう。

空間変数を使って2次元空間上に3次元空間を表現させる。具体的には、高低差のある地形図ファイルを読み込んで、マップに表す。エージェントの動きがその勾配によって左右される。具体的には、尾根づたいに登山するエージェントと沢づたいに登山するエージェントがいる。

1. Universeの下に空間mountain(200×200, 他はデフォルト)を作成します。
2. mountainの下にエージェントridge(尾根登り)とstream(沢登り)を追加します(それぞれ20ずつ作成)。
3. mountainの下に整数型変数elevationと実数型変数shadeを追加します(elevationは実際の標高, shadeは空間上に陰影を表現するための数値です)。
4. マップ出力をします。まずマップ出力設定で、原点位置を左上にしておいて下さい(これから取り込む画像データの設定上、こうしておかないと富士山が反転して表示されてしまいます)。次に要素設定で、各エージェントと、空間変数はshadeのみを追加、変数範囲は100から250までにします。色は任意ですが、なるべく山らしいものにしましょう。この変数は北西方向、仰角45度から光をあてたときの陰影の値を示すもので、標高の高さそのものを反映しているのではなく、値が高いほど光の当たっている場所を表しています。ここでは250を最も明るい色にしましょう。
5. 富士山の実際の標高データと、陰影を表現するためのデータを空間に入力します。手順は前章33.5を参照して下さい。山影研究室のホームページ<http://citrus.c.u-tokyo.ac.jp/artisotextbook>からダウンロードしてきた34materials.zipを解凍し、展開された34materialフォルダの中のファイルを使います。elevationには、fuji-elevation.csvファイルを選択します。同様にshadeもファイル入力して下さい。こちらはfuji-shade.csvファイルを選択します。これで富士山のデータがモデルに取り込まれました
6. モデル名をfujiとして保存して下さい。実行してみましょう。空間に富士山が現れたでしょうか?(図34.4参照)

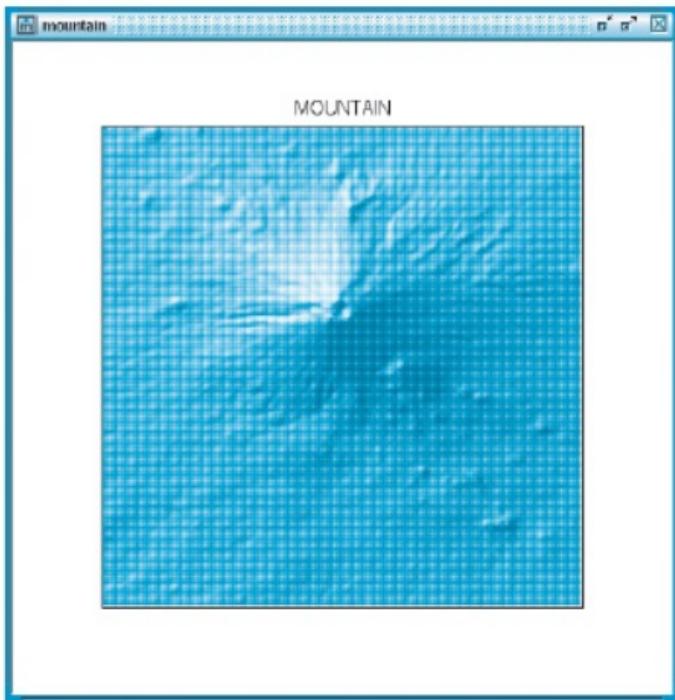


図34.4

ちなみに、自分でひとつひとつの座標に標高を入れていくことにより、好きな山を作成することも可能です。空間変数の初期値設定は、Dim1がX座標、Dim2がY座標となっており、各座標位置に合わせて変数名のカラムに数字を入れていくと、任意の傾斜で好みの山が作れます。時間があれば、試してみて下さい。

34.6 地形図の「勾配」をエージェントの行動に反映させる

次にエージェントルールに移りましょう。2種類のエージェントのルールは以下のとおりです。

- 尾根登りをするridgeエージェントのルール：

1. フォ文で自分の周囲8近傍空間へ移動し、その空間変数の値を取得して、また戻る。
2. もし今いる場所よりも高いところがあれば、その中で最も高い場所へ移動する
3. 高いところが1つもなく、かつ8近傍がすべて同じ値(平坦)だったら、ランダムな方向へ移動
4. 上記以外なら動かない

- 沢登りをするstreamエージェントのルール：

1, 3, 4はridgeと同じ

2. もし今いる場所よりも高いところがあれば、その中で最も低い場所へ移動する

つまり、ridgeはとりあえず高みを目指し尾根に行こうとするのに対し、streamは沢づたいに山を登っていく、というエージェントです。

ルールを下記のように記入しましょう。

◎ridge(尾根登りエージェント)のルール

```
Agt_Init{
    //ランダムに配置する
    My.X = Int(Rnd()) * GetWidthSpace(Universe.mountain)
    My.Y = Int(Rnd()) * GetHeightSpace(Universe.mountain)
}

Agt_Step{
    Dim i As Integer
    Dim slope As Integer
    Dim dx As Integer
    Dim dy As Integer
    Dim same As Integer
```

```
slope = Universe.mountain.elevation(My.X, My.Y, 0)
dx = My.X
dy = My.Y
same = 0

//8方向へ行って傾斜を確かめる
For i = 0 to 7
    ForwardDirectionCell(i, 1)
    //周りに高い場所があった場合、最も高い場所の座標を取得する
    If slope < Universe.mountain.elevation(My.X, My.Y, 0)
Then
    dx = My.X
    dy = My.Y
    slope = Universe.mountain.elevation(My.X, My.Y, 0)
//傾斜が同じときはその数をカウントする
Elseif slope == Universe.mountain.elevation(My.X, My.Y,
0) Then
    same = same + 1
End if
//戻る
If i < 4 Then
    ForwardDirectionCell(i + 4, 1)
Else
    ForwardDirectionCell(i - 4, 1)
End if
Next i
```

```
// 平坦だったらランダムに移動、より高い場所が近傍にあれば、目標へ移動
If same == 8 Then
    ForwardDirectionCell(Int(Rnd() * 8), 1)
Else
    My.X = dx
    My.Y = dy
End if
}
```

◎stream(沢登りエージェント)のルール

Agt_Init{ }のルールはridgeと同じ

```
Agt_Step{
Dim i As Integer
Dim slope As Integer
Dim slope2 As Integer
Dim cou As integer
Dim dx As integer
Dim dy As integer
Dim same As Integer

slope = Universe.mountain.elevation(My.X, My.Y, 0)
slope2 = Universe.mountain.elevation(My.X, My.Y, 0)
cou = 0
dx = My.X
dy = My.Y
same = 0
```

```

//8方向へいって傾斜を確かめる

For i = 0 To 7

    ForwardDirectionCell(i, 1)

    If slope < Universe.mountain.elevation(My.X, My.Y, 0) Then

        cou = cou + 1

    //周りに高い場所があった場合、最も低い場所の座標を取得する

    If cou == 1 Then

        slope2 = Universe.mountain.elevation(My.X, My.Y, 0)

        dx = My.X

        dy = My.Y

    Elseif slope2 > Universe.mountain.elevation(My.X, My.Y, 0)

0) Then

    dx = My.X

    dy = My.Y

    slope2 = Universe.mountain.elevation(My.X, My.Y, 0)

End if

//以下、ridgeと同じ

//傾斜が同じときはその数をカウントする

//戻る

//平坦だったらランダムに移動、より高い場所が近傍にあれば、目標へ移動

}

```

以上までの作業をfiji(B)と名付けて保存し、実行してみましょう。ridgeエージェントとstreamエージェントの登山の仕方の違いが観察できたでしょうか？また、山頂ではなく、小高い丘で止まってしまうエージェントもいるでしょう。しばらくモデルを眺めて、登山した気分になりましょう。

さて、上のルールは、かなり複雑なことをしていると感じたのではないでしょうか。なぜわざわざフォ文で周

りを見にいて傾斜を確かめてから、また戻るというルールにしたのでしょうか？ 実はこのようなルールにしないと、モデルが動かないのです。空間変数にはループというものがないため、たとえば座標(0, 0)の標高と、その左や下の（存在しない）座標の標高を比較しようとすると、エラーが出てします。そのようなエラーを出さないためには、上のモデルのように、1ステップの間に周りに進んで確かめてから戻ってくるという工夫が必要になるのです。このような技法は第11章で学びましたね。空間変数の末端処理には、注意することを心がけましょう。

（保城広至）

新しく学んだ事項

- エージェントの代用としての空間変数
- ループしない空間での末端処理への注意
- 値画面出力設定
- 外部ファイルを読み取ることで空間を三次元に見せる

第35章

実験のための道具を活用する

35.0 作ったあとも重要です

- シミュレーションに人間が介入できます
- まったく同じふるまいも再現可能です
- 亂数を知り、かつ使いこなしましょう
- 実行・終了をartisocに一任できます

35.1 作ったモデルを使いこなす

これまで、どのようにモデルを作るのか、ということを中心に説明してきました。しかし、ただ単にモデルを作つてそれらしく動けばよい、という場合ばかりではありません。対象を抽象化した「モデル」がどんな動きをするのか理解したり、面白い例や一般的な傾向を見つけたりしたいこともあるからです。つまり、モデルを作つたあとには、それを使った「実験」とその結果の分析が控えているわけです。

特にモデルを本格的な研究や実務に活用する場合、こうしたモデルを作つたとの作業が大きなウェイトを占めることになります。artisocは、モデルの作成をサポートするだけではなく、実験を行なう上で便利な機能をいくつも備えています。この章では、こうした機能を活用した手法をいくつか紹介します。

35.2 シミュレーション実行中に手を加える

モデルを実行していると、このタイミングでこんなことが起きたらどうなるだろう？と思うことがしばしばあります。たとえば、生態系のモデルで捕食者が絶滅してしまいそうなときに「あと一匹多くいたらどうなるだろう？」とか、交通渋滞のモデルで「いま一車線を閉鎖したらどうなるだろう？」といった状況です。もちろん、ルールによって自動化することもできますが、シミュレーションの状況を見ながら人間（モデル作成者）が操

作を加えることで、より柔軟に探しを入れることができます。

コントロールパネルのスライドバーを使って実行中のシミュレーションにユーザが介入する例は、既に第8章で出てきました。ルール記述を使うと、さらに介入の自由度が増します。使い方は意外に簡単です。**コントロールパネル設定** から、「ルール記述」を選び、ボタンを押したときに実行したいルールを書き込みます(図35.1参照)。一時的な変数を定義する事もできますし、エージェント集合の作成・操作、エージェントの生成・削除、コンソール画面への出力やファイル入出力に至るまで、Universeのルールエディタに書くと同じようにルールを書く事ができます。ただし、コントロールパネルはエージェントではないので、X, Y, Direction, Layerといった変数を持ちません。したがって、自分の方向や位置を変更するTurn()やForward()などの関数はUniverseのルールエディタの中と同様に使えません。

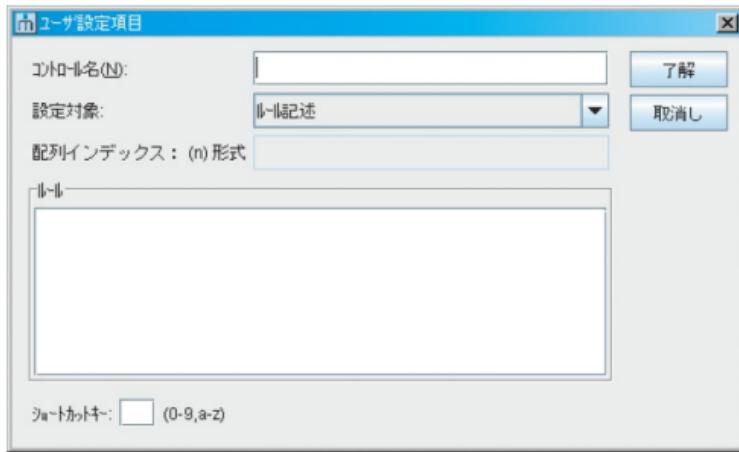


図35.1

35.3 亂数シード値を使って偶然の要素を排除する

artisocを研究や実務に利用していると、同じモデルで、パラメータ(変数)を変化させて、その影響を厳密に比較したい場面が出てくることでしょう。しかし、パラメータだけを変更したつもりでいても、他の条件

が全く同じとは限りません。その原因は乱数にあります。自分が進む方向や相互作用の相手を選ぶときにRnd()を使つていれば、それらは試行ごとに違つてしまふからです。さらに、たとえルールに明示的に書いていなくても、ステップ内でのエージェントの実行順序は乱数に基づいて決まっています(第31章を参照)。同期を取つていないモデルではこれによって結果が変わることがあります(第19章を参照)。

多くのモデルでは実行順序やランダムな選択が多少違つても、同じような結果が出てくることでしょう。しかし、中には「ブラジルで蝶が羽ばたいたせいで、テキサスで竜巻が起きる」ような、初期のちょっとした違いが、結果の大きな違いを生むモデルもあります。こういうモデルの結果は、パラメータばかりでなく、偶然にも大きく依存します。

では、偶然の要素を排した結果を得るためにには、どうしたらよいのでしょうか? ひとつの解決策は、大数の法則を使うことです。つまり、ある条件で何度も何度も実験し、どういう傾向の結果が出るかを調べるのです(この方法についてはあとで説明します)。しかし、何度も実験する方法では、「もし条件が違つたら、ある特定の一回の試行の結果がどのように違つたか」はわかりません。このような、いわゆる「歴史のイフ」を知りたいときには、同じ乱数列を用いて実験する方法があります。

数字が同じ順番で出てくるのでは乱数じゃないじゃないか、と思うかもしれません。たしかにその疑問はもつともです。本来、乱数とはサイコロを振ったときに出る値のように、予測不可能な値です。しかし、本当にランダムな値を作る事は意外に難しく、専用の機械を備えていない多くのコンピュータは、特定の規則に従つた計算によってバラバラな値を出して、乱数として使っています。これは擬似乱数と呼ばれます。artisocでこれまで使つてきたRnd()も実は擬似乱数です。

擬似乱数は、シード値と呼ばれる値を元にして、一様分布になるように計算された一見バラバラな数です。バラバラな数字ですが、計算によって作られているので、論理的にはそれまでの数列から次の数が予測可能ですし、シード値がわかれば、全く同じ乱数列を再現できます。この性質を利用し、シード値を固定することで、全く同じ試行を再現する事ができるのです。

指定の仕方は簡単です。「実行環境設定」ダイアログの「乱数シード値」の欄に、適当な数字を入力するだけです。空欄にしておけば、シード値は実行ボタンを押した瞬間の詳細な時刻データを元に生成

されますから、自動的に毎回違う乱数列が出てきます。後述する連続実行を行なうと、はじめの試行で使ったシード値に基づいた乱数列の続きをします。ただし、当然ですが、ルール内でRnd()が実行される回数が変わったり、エージェントの数が変わったりすると、同じ乱数が同じ場所で使われませんから、その点は注意が必要です。

乱数シード値には、もうひとつ便利な利用方法があります。GetRandomSeed()という関数を用いると、その試行で用いているシード値を取得できます。これを用いれば、上手くいった実行例や珍しい実行例を、ログを取る(第36章を参照)ことなく再現することができます。

では、試しに第20章で作ったgame-of-lifeのシード値を書き出してみましょう。変更箇所は、Univ_Finish{ }に、PrintLn(GetRandomSeed())と書くだけです。これで、シミュレーションが終了するときに、コンソール画面にシード値が出てきます。実行を再現するには、それをコピーしてシード値の指定欄にペーストして **了解** としたあとに、実行ボタンを押しましょう。

35.4 いろいろな乱数

ところでこの乱数ですが、実はartisocの中には、いくつかの種類が組み込まれています。一様乱数のRnd()はもうおなじみですね。0以上1未満の疑似実数値が同じ確率でランダムに現れます。0～0.1の区間の数字が出る確率も、0.1～0.2の区間の数字が出る確率も同じですから、イフ文を使って、ランダムに行動を決める時に使う事ができました。また、エージェントを空間上に均等に初期配置するときにも便利でした。

しかし、一様に分布していない初期値を設定したい事もあるでしょう。たとえば、身長、体重などといった正規分布に従うような値をランダムに与えたいときです。そんなときは、NormInv()を使うと便利です。平均170、標準偏差6のランダムな値(成人男子の身長)を作ってコンソール画面に出力してみましょう。新しいモデルを作ってUniv_Step_Begin{ }に以下のように書いてみて下さい。

```
Dim a As Double  
a = 0  
Do while a = = 0  
    a = Rnd()
```

```
Loop  
PrintLn(NormInv(a, 170, 6))
```

NormInv()の第一引数には0超1未満の一様乱数を入れます。そのままRnd()とすると0が出てくる可能性もありますから、上ではいったんaの中に入れて、0になったときには選び直しています。Do while文については、コラム「繰り返し文いろいろ」を参照して下さい。

実行してみて下さい。もっともらしい値が羅列されたでしょう。

別の乱数もあります。たとえば、1ステップを1分と考えた待ち行列のシミュレーションを考えましょう。あるATMコーナーには1分あたり平均1.6人のお客様が来ます。お客様をどのように生成したらよいでしょうか。37.5秒に1人ずつ生成することもできますが、ちょっと不自然すぎます。ある1分間の来客数を見たとき、一番多いのは1人か2人でしょうが、誰も来ないことも、ときには一度に4人来る事もあるでしょう。このような場合や、ある地区の一日の交通事故の件数など、一定時間内にその事象が起きる確率が低く、事象の発生が独立している場合に適している乱数の分布は、ポアソン分布と呼ばれています。この場合、PoissonRnd(1.6)とすることで、各ステップでのお客様の人数をランダムに、かつ、平均値に沿った形で求めることができます。ポアソン分布は、平均と分散が同じ値になるので、正規分布とは異なり、引数は1つだけです。また、NormInv()で求められる値が実数型なのに対して、PoissonRnd()では整数型が求められる事に注意して下さい。

実際のルールでは、Univ_Step_Begin{ }に

```
Dim i As Integre  
For i = 0 to PoissonRnd(1.6) - 1  
    CreateAgt(Universe.atm.customer)  
Next i
```

と書けば、お客様がステップ平均1.6人新たに生成されます。

ところでPoissonRnd()の値はゼロになることもあります。その場合、フォ文はどうなるのでしょうか。実は、エラーにならずにフォ文全体が無視されるので、何も実行されません。

35.3でも述べたように、乱数によってさまざまに偶然性が組み込まれたモデルでは、あるシミュレーションの試行で得られた結果が別の試行で再現されるとは限りません。こうした場合、シミュレーションの結果の再現性を検討したり、あるいはモデルのふるまいが示す一般的な傾向を抽出したりする必要が出てきます。シミュレーションを繰り返し実行し、出てきた結果を分析することが、そのための最も基本的な作業になります。

シミュレーションを繰り返し実行するにあたって、「手動」で実行ボタンと停止ボタンを何十回・何百回と交互に押し続けるのは、それだけでなかなか大変な作業です。さらに条件(パラメータ)を変えてシミュレーションを繰り返す場合、これに、コントロールパネルの操作という面倒も加わります。幸い、artisocにはこれらの操作を自動で行なってくれる連続実行の機能が備わっています。以下では、第13章で作ったinfection(B)モデルを例に取り上げて、この機能の活用法を紹介していきます。

まず、artisocでシミュレーションの連続実行を行なうための手順の概略をまとめておきましょう。

1. ルールの中か、「実行環境設定」で、1回のシミュレーションの終了条件を設定する
2. 出力する値や、出力の方法・タイミングを決めて、必要なルールを書き込む
3. 「実行環境設定」の「連続実行」タブで連続実行の条件を指定する

(1) 終了条件の設定

当然のことですが、一回のシミュレーションが延々と途切れることなく続くような場合、それを繰り返し実行することはできません。1回のシミュレーションが有限のステップ数で終わるように適切な終了条件を指定する必要があります。シミュレーションの終了条件の設定方法は、大別して二種類あります。ひとつは、イフ文とExitSimulationMsgLn()関数などを組み合わせて、モデルの内部(たとえばUniv_Step_End{})でルールとして指定するやり方で、「全員が満足したら終了する」「全てのセルの状態が変化しなくなったら終了する」など、これまで何度も用いてきた、既におなじみの方法です。もうひとつの設定方法は、「実行環境設定」ダイアログでシミュレーションの終了条件を書き込むことで、モデルの外部から終了条件を

操作する方法です。ここでは、この後者のやり方を取り上げましょう。

まず、infection(B)モデルを開いて、**設定** メニューから **実行環境設定** を選んでください。最初に出てくる **シミュレーション** タブの「シミュレーション終了条件」の部分に終了条件を書き込みます(図35.2参照)。終了条件は、「最大ステップ数」「最大時間」「終了条件式」の三種類の形式で指定できます。前二者は一回のシミュレーションの継続時間を、それぞれステップ単位・実時間の分単位で指定することができます。「終了条件式」には、終了条件をartisocのルールの形式(イフ文の条件式の形式)で書き込みます。たとえば、感染者の数がゼロになった場合、シミュレーションを終了させるには、ここに「Universe.number == 0」というふうに書けばいいわけです。

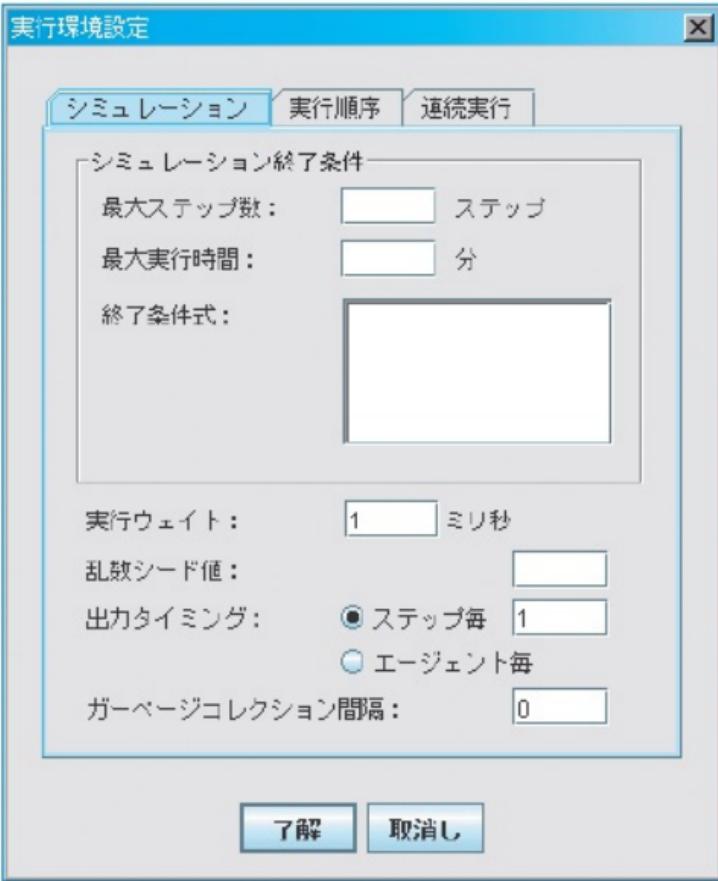


図35.2

以下では、1回のシミュレーションは100ステップで終了するものとし、これを繰り返し実行することにします。そこで、まず「最大ステップ数」の部分に「100」と入力し、**了解**を押しましょう。ファイル名をauto-runとして保存した上で実行します。指定したとおりに100ステップで終了するか確認して下さい。

(2) 出力のための準備

artisocでは、モデルで使う変数の値(たとえばUniverse.number)は、1回のシミュレーションが終わると全て初期化されてしまいます。シミュレーションが繰り返し実行されている間、ずっと画面に見入っているなら話は別ですが(これはこれで大変です)、シミュレーションを多数回にわたって実行する場合、注目している変数の値を何らかの形で出力し記録しておく必要があります。

連続実行の際にしばしば用いられる出力の方法として、シミュレーション毎の変数値のコンソール画面への書き出し、および出力設定のファイル出力やファイル出力関数を使用する外部ファイルへの書き出し、の二つがあります。本格的な研究に使われる後者の方の方法、特にファイル出力関数の使用法については、次の第36章で学ぶことになります。ここでは、より簡単に行なえる前者の方法を紹介しましょう。

コンソール画面には、PrintLn()を使って、これまででもシミュレーションの終了ステップ数などを書き出してきました。そのときに気づいたかもしれません、コンソール画面に記録された数値や文字は、何度シミュレーションを繰り返しても、artisoc本体を閉じないかぎり、画面上に書き残されたままです。そこで、これを利用して、連続実行時の変数の値を逐次コンソール画面に出力していくようinfection(B)モデルに変更を加えることにします(ちなみに、連続実行時には時系列グラフにも全試行の変数値の変化が連なって表示されますが、正確な値を求めるには適していません)。

ここでは、infection(B)モデルの1回のシミュレーション(100ステップ)が終了した時点で、PrintLn()を使ってコンソール画面に感染者の割合(Universe.number/Universe.pop)を書き出すことにします。第8章で学んだように、シミュレーション終了時に実行するルールは、Univ_Finish{ }に書き込みます(Univ_Step_End{ }に)出力ルールを書くとステップ毎に出力することができます)。Universeのルールエディタを開いて、次のようなルールを記述しましょう。

```
Univ_Finish{
PrintLn(CountSimulationNumber() & " --> " & Universe.number /
Universe.pop)
}
```

上記ルール中にあるCountSimulationNumber()は、連続実行時に、現在何回目のシミュレーション

ンが実行されているのかを知らせてくれる関数です。たとえば7回目のシミュレーションの最後に感染者の割合が0.75であったとすると、この試行の結果として、「 $7 \rightarrow 0.75$ 」がコンソール画面に書き出されることになります。

(3)連続実行の条件指定

ここまで準備ができたら、あとは連続実行の具体的な条件を指定するだけです。再び「**実行環境設定**」ダイアログを開きましょう。3つのタブから、「**連続実行**」を選んで下さい。すると新たな画面が開き、そこには「初期値変化設定」の3つのオプションが並んでいるはずです。各オプションの内容は次のようにまとめることができます。どれを選ぶかはモデルを使った実験・研究の目的に応じて変わってきます。

- 初期値変化「なし」: パラメータ(たとえばUniverse.popやUniverse.infiltratio)を固定したまま、「最大実行数」で指定した回数だけシミュレーションを実行する。
- 初期値変化「線形型」: 「初期値変化変数名」で指定したパラメータを、「開始値」と「終了値」で指定した範囲内で「変化幅」の分だけ漸増させていき連続実行を行なう。各パラメータの値で「同じ初期値の実行回数」だけ繰り返しシミュレーションをする。
- 初期値変化「ランダム型」: 「初期値変化変数名」で指定したパラメータに、「最大値」と「最小値」で指定した範囲内からランダムに選んだ値を代入して連続実行を行なう。「異なる初期値の実行回数」で異なるパラメータをどれだけ発生させるかを指定し、「同じ初期値の実行回数」で各パラメータの値でのシミュレーションの実行回数を指定する。エージェント直下の変数を指定して「同位の変数に異なる値を設定」にチェックを入れると、そのエージェントが複数存在する時に、エージェントごとに異なる値を代入する。チェックが入っていないときには、値はそのエージェント種の全個体で共通となる。

ここでは、二番目の「線形型」を使って、infection(B)モデルのpersonエージェントの数(Universe.pop)を100から500まで50刻みで変化させ、各popの値で10回ずつシミュレーションを行なうように連続実行の条件を設定してみて下さい。[図35.3](#)のように各項目を埋めていきます。



図35.3

以上で連続実行のための準備は完了です。ファイル名をauto-run(B)として保存しておきましょう。コントロールパネルで適当に初期感染率initratioを設定した上で、実行ボタンを押してみて下さい。90回のシミュレーションが連続して実行され、各試行終了時の感染者の割合がコンソール画面にリアルタイムに

出力されていきます。感染者の割合が激増を始める臨界的な人口数は、果たして存在するのでしょうか？

(阪本拓人・鈴木一敏)

新しく学んだ事項

- コントロールパネルの「ルール記述」
- 亂数シード値
- GetRandomSeed()
- さまざまな擬似乱数
- NormInv()
- PoissonRnd()
- 実行環境設定での終了条件の指定
- GetCountSimulationNumber()
- 連続実行のやり方

英語圏で使用する際の注意

他のマルチエージェント・シミュレータと比較すると、artisocは日本語環境で使用できるという大きな特徴を持っています。artisocは、OSの種類を選ばないだけではなく、日本語環境でも英語環境でも動作するきわめて汎用性の高いシミュレータです。ただ、どんなモデルでも、無条件にいつでもどこでも動かせるというわけではありません。

特に、artisocで作ったモデルのファイルを異なる言語環境（日本語版OSか英語版OSか、後者に日本語フォントは備わっているかなど）で用いる場合には注意が必要です。日本語が混じっていると、英語環境のartisocではモデルが正常に動作しない可能性があります。したがって、英語圏でのデモ（プレゼン）や研究・実務などを念頭において、日本語環境のartisocでモデルを作る場合には、以下の点に留意するようにして下さい。

- 空間名・エージェント名・変数名などを日本語（全角フォントも含む）で定義しない（エラーになります）
- 出力設定やコントロールパネルのタイトル、凡例なども日本語では書かない（文字化けで済みますが）
- コメント文も日本語では書かない（文字化けで済みますが）

要するに、「日本語（全角フォント）はいっさい用いない」という原則にしたがうのが、最も安全だということになります。

もっとも、モデルが完成するまでは、コメント文は日本語で書いておく方がわかりやすいです。

（阪）

第36章

ログ機能とファイル入出力関数を活用する

36.0 備えあれば憂いなし

- シミュレーションの「ログ」を記録して再生できます
- プрезентーションに大いに活用しましょう
- データをモデルに読む込むこともできます
- 結果をデータとして出力して、解析することもできます

36.1 ログ機能を使いましょう

ログとは、コンピュータを操作した際に残るデータ記録のことです。この記録を調べることで、そのコンピュータがどのような処理を行なったかがわかります。artisocはシミュレーションのログを記録して、再生する機能を備えています。ログを記録しておけば、同じふるまいを何度も再現させることができます。そのおかげで、たとえばプレゼンテーションを行なう際に、その場でシミュレーションを実行するよりも、ログを再生させる方が、描画や説明を容易に行なうことができます。任意のステップから再生させたり、逆再生を行なわせたりすることも可能です。また、非常に時間がかかるモデルを実行させるときは、画面をじっと眺め続けるよりも、ログをとっておいて、シミュレーションを終了した後でその記録を（適当に飛ばしながら）見たほうが便利ですね。ここでは、第20章で作成したコンウェイの「ライフゲーム」モデルを使って、ログを記録し、再生する方法を説明します。

36.2 ログを記録する

ログを記録することは、きわめて簡単です。まずgame-of-lifeモデルを開きます。**実行** メニューの**記録して実行** を選択すると、**シミュレーションメモ入力** ダイアログが現れます。ここではたとえば、logtestと名前を入れましょう（図36.1）。**了解** を押すと、通常と同じようにシミュレーションが実行され

ます。シミュレーションを終了させると、ログの記録も終了します。



図36.1

ちなみにログファイルは、モデルが置いてあるフォルダと同じ場所に自動的に作られるreplay logというフォルダにデータベースとして格納されます。replay logの中のファイルを直接起動しても、ログは再生されません。再生には、以下の作業が必要となります。

36.3 ログを再生する

ログメニューの「再生」を選択すると、シミュレーション選択ダイアログが現れます(図36.2)。記録したログは、「(モデルファイル名)-artisoc : (シミュレーションメモ)」という名前で記録されています。ログが複数ある場合は、プルダウンメニューから該当するものを選択します。「了解」を押すとシミュレーションの再生が可能な状態になります。「削除」を押すと、その記録は削除されます。



図36.2

ではlogtestを選択して、**了解**を押しましょう。

画面が変わったのが見て取れたでしょうか？まずモデル名に、「ログ再生」という表示がつきました。また実行パネルが再生パネルへと変わりました。左から順にボタンの説明をしていきます（[図36.3を参照](#)）。



図36.3

一番左のボタンは、「逆再生」です。最後から巻き戻しながら連続再生することができます。「ログ」メニューの「逆再生」を選択しても同じです。

左から二番目のボタンは、「再生」ボタンです。ログを最後まで連続して再生してくれます。「ログ」メニューの「再生」を選択しても同じです。

中央のボタンは、「ステップ再生」ボタンです。押すたびに1ステップ毎再生されます。

右から二番目のボタンは、「一時停止」ボタンです。ログ再生中に押すと一時停止します。「ログ」メニューの「一時停止」を選択しても同じです。

一番右のボタンは、「停止」ボタンです。これを押すことで、ログ再生を停止させます。「ログ」メニューの「停止」と同じです。

スライドバーは、クリック・アンド・ドラッグすることによって、表示させたいステップを自由に選択することができます。ログを200ステップ記録しておけば、左端が0ステップ、右端が200ステップとなります。

また、再生パネルの右端には、数字が出力されます。これは現在のログのステップ数を表すとともに、ここに直接数字を入力することで、指定したステップの状態を画面に表示させることも可能です。

36.4 ログをとることのメリット

今までさまざまなモデルを作成し、何回もシミュレーションを実行してきた人には実感できると思いますが、乱数を用いたモデルでは、同じルール、同じ初期配置、同じエージェント数でも、それぞれのシミュレーションの実行内容は毎回違ってきます。したがって、乱数を用いたモデルでは、全く同じふるまいを出現させるためには、二つの方法が考えられます。ひとつは乱数のシード値を固定すること(乱数のシード値は第35章を参照して下さい)、そしてもうひとつが、ログ機能を利用することです。

たとえばシミュレーションを実行中に、今までの状態が一瞬にして全く異なった状態に変わってしまったとき(このような現象のことを相転移と呼びます)、ログを保存しておけば、その前後の状態を詳細に観察することができます。また、シミュレーションを実演しながらプレゼンテーションを行なう場合、実行過程のストリーをあらかじめ知っていれば、解説も楽ですし、時間管理もきっちりとできます。逆にどのような結果になるかわからないシミュレーションをその場で実行すると、偶然にも自分の主張と正反対の結果が現れてしまったり、あるいはエラーが発生してシミュレーションが止まってしまう可能性がないともいえません。このようなことのないように、ログ機能を有効利用することをお勧めします。

36.5 ファイル入出力関数も使いましょう

シミュレーションを行なうとき、モデルによっては、変数の初期値などをモデルに読み込んで実験したいこともあります。artisocには、データをファイルから読み込む機能があります。また、シミュレーションの結果

(最終結果だけでなく、過程も)を統計ソフトで分析したいこともあるでしょう。artisocには、シミュレーションで得られたデータをファイルに出力する機能もあります。

artisocの内側と外側とでデータをやりとりするためには、ファイル入出力関数という特別の関数を利用します。これを使うと、簡単にデータをモデルに読み込んだり、データを外に書き出したりすることができます。

以下では、第10章で作成した分居モデル(schelling(C))を使って、ファイル入出力関数を使用する方法について学びます。

36.6 入力したいファイルを作成しておく

これまでの分居モデルでは、pennyとdim eの許容限度は、全てのエージェントの値について、3分の1で一定でした。ここでは、各エージェントが、異なる許容限度を持つ場合を考えます。ここでのモデルでは、異なる許容限度を記述したデータファイルをあらかじめ用意しておいて、それをモデルに読み込むという方法をとることにします。

分居モデルでは、45のエージェントがいたので、少し手間がかかりますが、0から1までの許容限度を45行記述した[図36.4](#)のようなテキストファイルを作成し、schelling(C)と同じフォルダにおいて下さい(コンマで区切られた45個の数字であれば、1行でもかまいません)。テキストファイルはinput.txtと名付けて下さい。なお、artisocの他の版(たとえばartisoc academic)では、任意のファイル名でよいのですが、artisoc textbookではinputというファイル名しか使えません。36.8で出てくるoutput.txtも同様です。

(1行)	0.5
(2行)	0
	0.4
・	1
・	0.4
・	・
・	・
・	・
(45行)	0.2

図36.4

36.7 外部ファイルをモデルに読み込ませる

ファイルを読み込むために使用する関数は、`OpenFileCSV()`, `ReadFileCSV()`, `CloseFileCSV()`の3つです。ファイルからデータを読み込むルールは、次のような手続きで行なわれます。

1. ファイルを開く:`OpenFileCSV("ファイル名", ファイル識別番号, 操作の種類)`

2. 開かれたファイルを読み込む : ReadFileCSV(読み込むファイル識別番号)
3. ファイルを閉じる : CloseFileCSV(閉じるファイル識別番号)

ファイルのデータを読み込むためには、まず「ファイルを開く」という作業が必要です。1のOpenFileCSV()の中には、まず、最初に開きたいファイル名を指定し、2番目の値として、ファイル識別番号というものを指定します。ルールの中では、実際のファイル名ではなく、全てこのファイル識別番号でファイルを操作します。ファイル識別番号の値は何でもかまいませんが、複数のファイルを同時に開く場合には、別々の番号を付ける必要があります。3番目に入力する「操作の種類」は、どのような操作をするためにファイルを開くのかを指定します。1と指定すればファイルの中にあるデータを読み込むためにファイルを開きます、また、2と指定すれば、ファイルにもともとあったデータを全て破棄して、ファイルにゼロからデータを書き出すために開くことになり、3と指定すれば、ファイルにもともとあったデータを保持して、最後の行の下に追記するかたちで、データを書き出すために開くことになります。

OpenFileCSV()で開いたファイルのデータは、2のReadFileCSV()という作業によって読み込みます。ReadFileCSV()を1回実行すると、1行(あるいはコンマで区切られたデータを1つ)だけ読み込みます。したがって、今回のようにエージェントの数だけデータを読み込ませたい場合は、ReadFileCSV()を45回実行する必要があります。データはファイルの1行目から順番に読み込まれます。

開いたファイルは、必要な作業が終わったら、3のCloseFileCSV()を実行して、閉じるようにします。いつたん閉じたファイルを再び開いて、ReadFileCSV()を実行すると、また1行目から読み込まれます。したがって、45のデータを読み込むためには、45回の読み込み全てが終わってから、ファイルを閉じるようにします。

それでは、モデルを修正しましょう。

各エージェントに個別の許容限度を与えるために、10章で作成したschelling(C)のpennyとdimeの各々に、実数型変数limitを追加します。各エージェントの許容限度は、エージェントを生成した際にそれぞれに与えることにします。このモデルは、file-in-outと名付けて保存しましょう。

Univ_Init{ }に次のルールを追加して下さい。

```
Dim pd As Agt //冒頭に追加。
```

MakeAgtsetSpace(coin, Universe.chessboard)の下に、次のルールを追加。

```
OpenFileCSV("input.txt", 1, 1)
For each pd in coin
    pd.limit = ReadFileCSV(1)
Next pd
CloseFileCSV(1)
```

ここでは、ファイル識別番号を1としています。

また、ここで許容限度は、ファイルから読み込まれた値に従いますから、pennyとdimeのAgt_Step{ }の中のIf rate < 1/ 3 Thenを

```
If rate < My.limit Then
```

に変更してください。

このモデルを上書き保存して実行してみて下さい。もし読み込みファイルのなかの許容限度が1に近い値が多ければ、なかなか均衡しないはずです。逆に許容限度が0に近ければ、あっという間にシミュレーションは終了してしまいます。

36.8 実行結果を出力する

次に、モデルの実行結果をファイル出力してみましょう。ここでは、各ステップの分居度と満足しているエージェント数を出力することにします。これらの値は、**設定** > **出力設定** > **ファイル出力** を設定することでもファイル出力することができます。

ここでは、練習として、ファイル出力関数を使うことにします。

各ステップの結果を出力するので、Univ_Step_End{ }の冒頭に、次のルールを書きます。

```
OpenFileCSV("output.txt", 2, 3)
```

```
WriteFileCSV(2, GetCountStep(), False)
WriteFileCSV(2, Universe.averagelevel, False)
WriteFileCSV(2, Universe.satisfied, True)
CloseFileCSV(2)
```

ルールは、ファイル読み込みの場合とほとんど同じです。ファイルを開くルールでは、まず、書き出すファイル名output.txtを指定しています。次に、ファイル識別番号を指定します。ここでは、許容限度を読み込むファイルを既に閉じているので、識別番号は1でも問題ありませんが、とりあえず区別するために、2と指定しています。3番目には、操作の種類を指定しています。ここでは、各ステップの結果を書き出していくので、追記モードの3を指定しています(2と指定すると、それまでのデータを全て破棄して新しく記録していくので、最後のステップだけ記述されることになります)。

次に開いたファイルに書き出す作業を行なっています。WriteFileCSV()では、最初にファイル識別番号(ここでは2)を指定し、2番目に、出力する内容を指定します。上のルールの3つのWriteFileCSV()のうち、最初のWriteFileCSV()では、現在のステップ数を書き出すので、GetCountStep()関数を実行しています。また、3番目には、出力する値を書き出した後に、改行を行なうかどうかを指定します。Trueとすれば改行し、Falseとすると、改行せずにコンマ(,)を追加します。なお、出力された値はそれぞれ" "で囲まれます。

したがって、上のルールでは、「"現在のステップ","分居度","満足しているエージェント数"(改行)」で1行が構成されています。各ステップでの書き出しが終わったら、CloseFileCSV()で、ファイルを閉じます。

file-in-out(B)と名付け保存してから、実行してみて下さい。実行が終わると、モデルファイルと同じフォルダの中に、output.txtというファイルが作成されているはずです。これを開くと、たとえば、次のような結果(以下では最初の5行のみ例示)が出力されているはずです。

```
"1","24.20119047619048","24"
"2","25.553571428571423","28"
"3","25.57142857142857","25"
"4","27.476190476190485","30"
"5","28.478571428571435","32"
```

これは、CSV形式のファイルと呼ばれるもので、データがコンマで区切られています(Comma Separated Valuesの略です)。ファイルの拡張子をtxtからcsvに変更すれば、そのままExcelなどの表計算ソフトに読み込むことができます。したがって、シミュレーション結果を統計分析することも可能ですね。

(保城広至)

新しく学んだ事項

- ログの記録と再生
- 外部データの入力
- 外部データとして出力
- ファイル入出力関数OpenFileCSV(), ReadFileCSV(), CloseFileCSV(), WriteFileCSV()

第37章

シミュレーションを疑ってみる

37.0 謙虚さが飛躍に繋がります

- ミスを減らすだけでは不十分です
- 面白い結果が正しい結果とは限りません
- 疑ってみる態度が必要です
- 限界を知っておくことも大切です
- 健全な懷疑に新しい成果が宿ります

37.1 疑う余裕を常に持とう

自分の作ったモデルでシミュレーションが実行でき、その結果が出てくれば、誰でもうれしいものです。特に、予期した結果が出ればもちろんですが、予想もしなかった結果が出てくるのも楽しいものです。趣味や娯楽でマルチエージェント・シミュレーションを実行しているだけなら、うれしさや楽しさだけで十分シミュレーション(趣味レーション?)をした甲斐があります。

しかし研究や実務のためなら、ここでいったん立ち止まることが大切です。シミュレーションをいろいろな角度から再考してみる(つまり、疑ってかかる)のも必要です。学会で報告したり、論文を発表したりしたあとで、結果や解釈に誤りがあったことがわかるかも知れません。シミュレーションの結果に基づいて政策やビジネスの意思決定をしても、現実の状況はシミュレーションの前提と大きく異なるかもしれません。シミュレーションの結果を鵜呑みにする前に、やるべきことが残っています。

この章では、公の場に人工社会を披露する前に注意すべき点についてまとめてあります。ポイントは、自分の成果をなるべく客観的にまとめた(冷めた・醒めた)視点で見直すことです。思い込みが強いと、うっかりミスを見逃すだけでなく、深刻な失敗を引き起こしかねません。過度に懷疑的になっては非生産的です

が、ほどぼりを冷まして吟味することが大事です。

37.2 ミスを疑う

モデルを作っている最中にエラー表示が出たり、実行ボタンを押したらエラー表示が出たりすると、いろいろするし、がっくりするものです。しかし、それは明々白々のミスであり、ある意味では実害はありません。自分のミスを疑うのは、エラーが出なくなり、シミュレーションの結果が出るようになってからが正念場です。

実行ボタンを押してから停止ボタンを押すまでシミュレーションが順調に進むとほっとします。予想しているとおりにシミュレーションが実行されるのもうれしいですが、予期しない結果が出ても「これこそ複雑系の神髄だ」と小躍りしたくなるものです。しかし結果が出たことで気を緩めてはいけません。ルール記述に際して、演算子を誤用していたり異なる変数名を混同しているかも知れません。たとえルールが正しく書けても、ルールやさまざまな設定に対するartisocの解釈がモデル作成者(つまり、あなた)の意図とちがってしまうのも、広義のミスです。第32章でまとめてあるように、自分が作ったモデルが実際にどのような動きをしているのかを詳しくチェックするデバッグ機能がartisocにはいろいろと備わっているので、是非活用して下さい。特に、「何か変だな」と感じるときにはもちろんですが、うまくいったと思える場合にもデバッグ機能を利用することを勧めます。

因みに、バグ(bug)とはプログラムの中に住みついで不具合を引き起こすムシのことで、ムシの除去を「バグ捕り」とか「デバッグ」といいます。デバッグとはそのための道具の事です。専門家によると、複雑なプログラムからバグを完全に除去することは不可能で、だんだん見つかなくなってしまっても、どこかに潜んでいると覚悟しておいた方がよいのだそうです。ということは、モデルの中から駆除できても、モデルよりはるかに複雑なartisocの中のどこかにまだ潜んでいるのかも知れません。私たちはさまざまな使い方をしてartisocの中のムシ捕りに励んできたので、普通の使用法に沿っている限りは問題ないと信じています。

37.3 モデルの構造を疑う

シミュレーションがモデルに従って「正しく」実行された、としましょう。シミュレーションが正しく実行されれば、結果が出ます。それが「正しい」結果であるのはもちろんです。しかし、その結果は、そのシミュレーション

ンが実行された条件に依存した、たまたまの結果です。このことはよく知られていることで、だから「何回も試行しろ」と言われています。しかし、何回も実行した後で、結果の平均(と分散)を計算すればよいというものではありません。

これからは、ミスやバグの問題ではなく、「正しい」モデルの中身を疑う段階に入ります。それは大別して、モデルの構造を吟味することと、モデルの挙動を吟味することです。これらは互いに結びついているので、両者の間を行き来しながら、モデルに対する信頼を高めていかなくてはなりません。

まず、モデルの構造を疑ってみましょう。

そもそもモデルの構造はどのように決まったのでしょうか。おそらく、理論とか仮説とか呼ばれる命題が既に提示されていたり、システムと呼ばれるような認識枠組みが既に作られていたりして、それをartisocに「移植」ないし「翻訳」しようとする過程で、モデルの構造が決まっていったのではないでしょうか。

私たちが現実世界を認識すること、認識対象をシミュレーションにまで繋げることの意味については次の最終章で扱います。ここでは、モデルの構造は私たちがモデルに反映させたい現実世界についてのアイデアを何らかの意味で具体化したものであることを確認しておけば十分でしょう。

経済学のように理論は数式で表現されるのが普通の場合もあれば、社会学や政治学のように自然言語が用いられるのが普通の場合もあります。どちらの場合でも、モデルの構造に変換する作業が必要です。前者なら、連続的な微分(方程式)の世界から離散的な差分(方程式)に変換し、さらに時間や時刻の扱いについても意識しなくてはなりません。後者の場合は、表現に曖昧さが残っていたり、多様な解釈を可能にしたりする部分を、明確な形で書き表さなくてはなりません。

このようなことを踏まえながら、モデルを作る際、エージェントについての仮定、相互作用についての仮定、環境についての仮定など、さまざまな仮定を置いたはずです。シミュレーションの結果を踏まえて、そのような仮定を見直しましょう。問題の本質から離れている部分の仮定は大胆に単純化し、重要だと思われる部分については慎重に検討する必要があります。

エージェントの行動ルールについても注意を払って下さい。そもそもエージェントという主体自体が、現実

の意思決定・行為主体を抽象化・単純化したものです。その意味では、エージェントは特定の機能しか持たない存在です。その特定の機能が問題の本質を捉えているのかどうか、が重要なのです。

皆さんの中には、人工社会の手法を導入して新しい理論を提唱しようと意気込んでいる人もいるでしょう。そのようなときには特に、既存理論とモデルの構造との関連づけを意識しながら、新しい理論にふさわしい人工社会モデルを構築して下さい。

37.4 モデルの挙動を疑う

モデルの挙動(つまりシミュレーション結果)は構造が生み出した「見える」部分です。次に、これを疑ってみましょう。人工社会はさまざまな偶然に左右されます。artisocでモデルを作るとき、ルールの中で一様乱数Rnd()を多用してきました。初期設定も偶然の要素が強く、シミュレーションの途中でもさまざまな偶然が介在します。また、モデルにはいくつかのパラメータが組み込まれていますが、パラメータの値の違いは結果に影響を及ぼします。

脆弱なモデルとは、このようなモデルに内在している不確定な要素の違いが、シミュレーション結果を大きく変えてしまうようなモデルです。他方、頑健なモデルとは、不確定な要素が大きく変わっても、結果にはあまり大きく響かないモデルです。この頑健性・脆弱性を調べることを感度分析といいます。つまり、モデルの感度(敏感性)をチェックするのです。

感度分析は、初期設定やパラメータの設定をいろいろに変えてモデルの挙動を調べることです。複雑なモデルになればなるほど、不確定な要素は増えますから、その組み合わせは累乗的に増えていくので、不確定な要素をばらばらに設定すると、どの違いが結果の違いを生み出したのかはっきりしない事態が生じます。ある特定の要素のモデルへの影響度を調べるには、他の条件を一定にする必要があります。要するに、感度分析は体系的に行なう必要があるのです。ここで力を発揮するのが、第35章で紹介した、乱数シード値や連続実行といったartisocの機能です。

初めは、あり得ないような極端から対極的な極端までの間を粗く区切って、モデルの大まかな特徴をつかみます。だんだんと、モデルの敏感な部分がどこかを絞っていき、敏感な部分については、設定を細かく

して、モデルの挙動を注意深く調べます。

たとえば、第11章のターミナルのモデルでは通勤客の大きな流れが現れました。このモデルでは、前方の対向客の存在を調べるルールは正面と左右45度、距離1、範囲1に設定されていますが、大きな流れができるのはこのような組み合わせの時だけかも知れません。そこで、角度や距離や範囲をいろいろに変える、エージェントにランダムな値を与えるなど、さまざまな条件でシミュレーションを繰り返しても大きな流れができるようなら、モデルは頑健だといえるでしょう。

前節で述べたように、モデルの構造の検討とモデルの挙動の検討とは関連しています。両者を結びつけて、モデルについての理解と自信を深めて下さい。

37.5 「よい」モデルかどうか疑う

多くの設定に対して脆弱なモデルは扱いにくいモデルです。シミュレーションの結果をまとめるためには、さまざまな条件の組み合わせごとに、何回も試行する必要があります。他方、頑健なモデルのシミュレーション結果に対しては、だいたいそんなものだろうと信頼度が高まります。頑健なモデルは、結果が安定しているだけに、さまざまな状況に適用できそうです。実際、頑健性でモデルを評価する場合があります。

それでは頑健なモデルがよいモデルで、脆弱なモデルは悪いモデルなのでしょうか。必ずしもそうではありません。感度分析によって絞り込まれた敏感な部分こそが、モデルによって把握したい不安定な現象に対応しているのかも知れません。臨界現象や相転移をモデル化しようとしている場合は、敏感な部分こそがモデルの中核です。

たとえば、第13章で作った流行モデルや、第19章の練習問題に課題として出した森林火災モデルでは、病気や火事が、人口や木の密度のある値を境に、下火になる傾向と蔓延(延焼)する傾向とにはっきりと分かれることができます。むしろ、そのような特徴を備えたモデルが、分析に必要とされているモデルです。

つまり、モデルの「よしあし」は、頑健かどうかとは別問題です。もちろん、「脆弱な」モデルが生み出すシミュレーション結果の「ぶれ」が現実とかけ離れているとか、モデル化の本質的な部分と関係が薄いとか、

モデルの見直しが必要な場合はもちろんあるでしょう。しかし、モデルがいくら頑健でも、頑健故に「不安定な現実」を反映していないなら、そんなモデルをこれ以上使っていてもあまり意味はありません。要するに、モデル作りの目的は頑健なモデルを作ることではなく、注目している現実の対象の特徴を「うまく」抽出しているモデルを作ることです。

自分の捉えたい現象が本当にモデルによって再現されているのかどうか、つまりよいモデルができたかどうか、という問題は、対象となった現象の抽象化とモデルの特徴との対応関係をさまざまな角度から吟味する必要のある難しい問題なのです。

37.6 現実との対応を疑う

一般的に言って、人工社会は抽象度が高いために、モデルやシミュレーション結果が現実世界にどのように対応しているのか判然としない場合が多くあります。artisocの空間の大きさ、エージェントの移動距離や視野の広さ、1ステップの間に実行される行動ルールなどを、実際の社会の大きさ、主体の行動半径、時間の流れなどと対応づける作業は必ずしも容易ではありません。

しかし、現実との対応が全くつかないモデルでは、シミュレーション結果をどのように解釈したらよいかもはっきりしないでしょう。人工社会はもともと抽象的に構築された社会ですが、そこにはモデル作成者にとって抽出したい社会現象の本質が組み込まれている必要があります。

たとえば、この本で何回か取り上げたトマス・シェリングの分居モデルは、それ自体はきわめてインパクトのある研究成果に繋がりましたが、特定の市街の分居状況がなぜ今あるような状態になったのかを調べるには無力です。市街の大きさ、道路や建物の配置、初期時点（たとえば30年前）における居住実態、地価や家賃、個人所得、人口動態、公共施設、法制度などさまざまな要因を組み込んだモデルが必要となります。

シミュレーションの対象となる現実世界の抽象度が下がるほど、現実との対応の程度は大きな問題となります。実際、学問分野によっては、モデルと現実との対応関係をはっきりさせないと説得力がない場合もあるでしょう。幸いなことに、人工社会の手法がいくつかの分野で徐々に受け入れられ始めたよう

す。それぞれ関心のある分野で、モデルやシミュレーション結果が現実世界とどのように結びつけられているのか、既存の研究例を調べてみるとよいでしょう。

もっとも、現実に近ければ近いほどよいのか、という疑問は残ります。これは人工社会の問題というよりは、モデル化一般の問題です。モデルを現実に近づけようとすればするほど、モデルは複雑になる傾向があります。一般的には、モデルはなるべく単純な方が好まれます。モデル作りの基本方針のひとつに、節約(parsimony)があります。辞書で調べると「吝嗇」というあまりよくない意味のようですが、ここでは、余計な無駄を省いてできるだけすっきりとさせるという意味です。「オッカムのカミソリ」ともいわれています。

抽象的な言い回しですが、一方では問題の本質についての抽象度が高く単純なモデル化を意識しつつ、他方では具体的な現実の問題との対応関係を意識してモデルを複雑にしていきましょう。個人的には、「必要最小限」な程度に複雑化すべきだと思いますが、いろいろな考え方があるようです。

37.7 応用は慎重に

理工系の学問分野で発達してきたシミュレーションの主要な目的は、シミュレーション結果をモデルの対象となった現実世界に「戻すこと」です。「戻し方」は予測、訓練、予備実験などさまざまです。その分野では、長い年月をかけ、さまざまな試行錯誤を重ねて、シミュレーションと現実世界との関係のつけ方を洗練させてきました。一方、人工社会については、研究の歴史が短いだけでなく、複雑な社会の単純なモデル化という側面を強調する研究が先行したために、モデルと現実世界との対応を明確にし、シミュレーション結果を現実世界に「戻すこと」についてはあまり関心を払っていませんでした。

しかし、たとえば第11章のターミナルのモデルを実在の駅の構造に近づければ、通勤客がスムーズに乗り換えられる改札口の位置やコンコースの設計に役立てることができるかも知れません。実際に最近になり、現実世界に「役立てる」ことをめざした使われ方もされるようになっています。たとえば、火災が発生した場合の地下街やビルから人々を安全かつ迅速に避難させる方法の模索、逆にパニックになった人々が安全に避難できる通路や非常口の設計、コンビニの売り上げを伸ばすために入店者の円滑な流れを考慮した商品の並べ方、津波警報が出た場合の地域社会にとっての最適な避難誘導法などです。いずれもエージェントの行動の相互作用を考慮したシミュレーションで、人工社会にふさわしい応用例といえ

るでしょう。しかし、人工社会のシミュレーション結果を現実世界に「戻す」方法は未発達で、まだ開発途上にあります。

人工社会は所詮モデルであり、コンピュータのなかの世界であって、私たちが対象としている世界そのものではありません。シミュレーションから得られた知見を現実の世界に反映させるには慎重さが必要です。現実との対応関係がはっきりしているモデルでさえ、そのモデルには人間が想定した要素しか組み込まれていません。当然ですが、想定外の要素について、シミュレーションは何も教えてくれません。このことは、人工社会にのみにあてはまるのではなく、シミュレーション一般について言えることですが、応用の歴史が短い人工社会については、特に強調しておくべきでしょう。人命に関わる場合はもちろん、高い社会的コストが発生しかねない現実世界への応用には慎重な態度で臨むことが基本中の基本です。

37.8 健全な懷疑に支えられて人工社会は発展します

この章では、モデルを疑うこと、シミュレーション結果を疑うことの重要性を述べてきましたが、人工社会に対する消極的な見方を広めることが本旨ではありません。むしろ逆に、人工社会が生み出す成果に自信を持つこと、そして成果を説得的に説くことの前提に、この章でまとめたような多角的に疑う姿勢があると思うからです。

ライト・シミュレータの最大の価値は何でしょうか。それは操縦に失敗しても誰も死なないということです。同様に、人工社会の中で戦争を防げなくとも、大飢饉を防げなくとも、そのかぎりにおいては「実害」はありません。シミュレーションの失敗を恐れる必要はありません。生じ得る最大のコストでさえ、artisocの強制終了による、保存していないモデルの消滅程度のことです。ここまで学んできた皆さんは、人工社会を「食わず嫌い」で否定することはないでしょう。失敗にめげず、失敗から学び、人工社会に取り組んで下さい。

新しい方法である人工社会は、これからも批判（あるいは無視）されるかも知れません。そのような反応（無反応）への最良の対処は、私たち自身の健全な懷疑にもとづいた成果を出し続けることでしょう。

いうまでもなく、人工社会は万能ではありません。しかし、従来の社会へのアプローチである観察者・分

析者の直感や思考実験が、人工社会より信頼できる結果を産むかどうかは大いに疑問です。長年の伝統だけに、根拠のない信用に支えられているだけかも知れません。たしかにシミュレーションには限界があります。しかし社会現象の理解を深める上で、人間による人間の省察という方法はそろそろ限界に近づいているのではないでしょうか。シミュレーション以外の方法がシミュレーションに勝るという一般的な根拠は示されていません。

モデルによる明確化とシミュレーション結果の分析という人工社会の方法は、いまだ不完全な理解に止まっている人間社会について、新しい理解をつけ加えることが可能な、有望な方法です。

いよいよ次章がこの本の最後になります。人工社会の作り方という技法的な観点から離れて、人工社会の可能性について考えてみたいと思います。

KISSしたら「ヤッコー」と蹴りを入れられた？

やたらにKissしてはいけないという教訓です。

シミュレーション研究の分野での理解では、KISSとはもともと軍隊(たぶんアメリカ軍の)用語でKeep it simple, stupid !(ゴチャゴチャやるな、ばかもんのことだそうです。もっとも異説もあって、Keep it short and simpleとかKeep it simple and smartという文章の省略形だそうです(andはどこにいった、とツッコミを入れたくなりますが、いずれにせよ、往年のロック・バンドとは無関係です)。要するにモデル作りは「なるべく単純に」という指針です。シミュレーションというと、現実(=本物)とは異なる「偽物」なので、なるべく現実に似せようと努力する姿勢も分からぬではありません。しかし複雑な現実(=森羅万象)に似せる努力は、往々にして「何が何だかわからない」結果になってしまいます。そこで、KISS指針でモデルを作りなさい、ということになるわけです。

それに対して、ヤッコーとは、「ヤッてみたらコーなった」という文章の省略形だそうで、ヤッホーとは関係ないようです。要するに、現実とかけ離れた単純なモデルを動かしてみて、「ヤッてみたらコーなった」と結果を見せるることはくだらない、という批判です。つまり、So What ?というわけです。だから、ヤッコーと言われたらヤッコーと言い返す、ということではダメなのです。

ではどうしたらよいのか。

社会現象のシミュレーションをやる場合に、ここが難しい問題です。KISS指針もヤッコー批判も、どちらもそれなりに重要なことだからです。やはり、どのような問題(=分析対象)を取り上げるのかを自覚して、その本質にふさわしいモデルを構築してシミュレーションするべきである、という正道を歩むしかないでしょう。

有名な笑い話があります(さまざまなバージョンがありますが)。

P:ここで何をしているんですか?

S:財布を落としちゃったので捜しているんです。

P:見当たらないですね。どの辺に落としたんですか?

S:あそこら辺だと思うんですが.....

P:じゃ、どうしてここで捜していたんですか!?

S:ほら、ここは街灯があって明るいから捜しやすいでしょ。

第38章

人工社会の対象と方法を俯瞰する

38.0 人工社会には大きな可能性が広がっています

- 次のステップへの序説です
- 距離を置いて、人工社会の発想を考えてみます
- 抽象的・一般的な論点を整理します
- 何らかのヒントになれば幸いです
- これからは、自由に羽ばたいて下さい

38.1 対象と方法

この本では、人工社会の作り方を、artisocに即して紹介してきました。

作り方とは要するに道具の使い方ですから、その伝授は、簡単な技法から複雑な技法へという段階的な部分と、さまざまな道具を使ってみせる並列的な部分とがどうしても交錯せざるを得ませんでした。また、さまざまな技法を細かい単位で、しかもなるべく完結するように試みたので、個々のモデルは小さいものにならざるを得ず、中には人工社会らしからぬモデルも登場しました。こうして学んだ多種多様な道具と技法を必要に応じて適当に組み合わせることにより、本格的な人工社会を構築することができます。

そこで、この本の締め括りとして、本格的な人工社会を構築する上でのヒントになるような話題に触れておくことにします。ひとつは、人工社会の対象となる私たちの社会、現実の社会の捉え方についてです。もうひとつは、人工社会の方法、すなわちモデルの構築とシミュレーションの実施の捉え方についてです。最後に、人工社会という新しい発想を適用するのにふさわしい状況(つまりコスト・パフォーマンスがよいと思われるテーマ)を並べてみます。

38.2 ムラ・クニ・地球、そしてその間隙

子供の頃よく読んでもらったおとぎ話によると「おじいさんは山へ柴刈りに、おばあさんは川へ洗濯に」でかけていったのに、おじいさんは女の赤ちゃんが詰まっている竹を見つけたり、おばあさんは男の赤ちゃんが入っている桃を拾ったりして、話はどんどん本筋(?)からずれていくことになるわけです。これは、個人的目的行動とその帰結が一致しないことがあるという教訓です。

現実には、そんなふうにして赤ちゃんに出会うことは稀です。おじいさんは誰かが先に柴を刈り取ってしまっていたので手ぶらで帰ってきたり、おばあさんは桃が流れてくるはずの水が上流でせき止められていて洗濯できなかったりすることの方がずっとあり得るでしょう。目的の結果が得られないということは、稀な突發的事象に出会うより、同じ村の他人の行動から影響を受けて生じる場合の方が圧倒的に頻繁です。

村人が柴を刈る入会地(共有地)や洗濯だけでなく飲み水や料理にも使う水を汲む(場合によっては灌漑にも使う)川には、その利用法について村人たちが守るべき制度があるのが普通です。しかし、自分だけ先駆けして柴を刈るとか山菜やキノコを探ると独り占めできます。たしかにそのときは独り占めできるかも知れませんが、やがて他の人も勝手なことをやり始め、結果として「元の木阿弥」どころかもっと悲惨な結果になってしまいます。みんながルールを守っているときに自分がルール破り(抜け駆け、ただ乗り)をする方が得になる状況の下でルールを守らない人が増えると「共有地の悲劇」と呼ばれる悲惨な結果を招くこともあります。

実際には、互いに顔見知りで、昔から世代を超えてつき合い続けてきた村落では、撻破りはあまり起こりません。村は、慣習・伝統・しがらみ・因習などを共有することによって規範・規則が比較的に守られているコミュニティ(共同体、地域社会)の典型例として考えられています。企業や大学のような大きな組織・団体も、ときどき「ムラ」と表記して比喩的に共同体に擬されることがあります。「ムラの撻」と呼べるような制度は、実際、いたるところに見られます。

大きな集合体の典型例は「クニ」でしょう。国(國)とも邦とも書きますが「クニ」は統治機関としての国家でもあり、人々が生活する国土でもあります。アイデンティティ(帰属意識)の核になっている場合もあるでしょう。今日、国家の主権者としての人間集団は、国民と呼ばれる共同体です。日本国民とかアメリカ

国民とか、国名を冠した共同体がひとつの大きなまとまりであることには間違いありません。因みに、唐代の即天武后が考え出したという「匱」という字（八方に広がるクニ）は、さらにスケールの大きな社会を思い浮かばせますね（だから、水戸黄門こと徳川光圀は諸国行脚をした?）。

近年、国際社会（インターナショナル・コミュニティ）とか地球社会（グローバル・コミュニティ）とか、国家よりもさらにおおきな集合体をひとつの社会ないし共同体として捉えようとする見方も登場しました。この他にも、宇宙船地球号とか地球村とかいった比喩を用いて、地球規模でひとつのまとまった人間の集合体を考えようとする立場があります。

いずれにせよ、数十人単位の集合体から数十億人単位の集合体まで、私たちはさまざまなレベルの社会ないし共同体として捉えることが可能です。それらはムラ・クニ・地球という具合に階層的に整序されているものもあるでしょう。しかし、多くの社会・共同体は、脱領域的に拡散していたり、そしてさまざまなタイプの集合体が、ある場合には入れ子状に、ある場合には重層的に重なり合ったり、またある場合には相互に競合したり、排他的な関係にあったりと、複雑に関係し合っています。

38.3 システム・モデル・シミュレーション

私たちがその中で生活している社会(共同体)を認識・分析・理解する方法にはさまざまなものがあります。そのひとつがシステムとして捉える方法です。交通システム、電力システム、流通システムといった社会の具体的・物理的な側面を抽出したり、親族システム、政治システム、金融システムといった社会のシンボルや価値観に関わる抽象的な側面を抽出したりと、社会のさまざまな捉え方が可能です。

システムという名前が明示的についていなくてもシステムとして捉えている場合も多々あります。たとえば、
じょう いちば
経済学にとっての市場がその典型です。それは、個々の市場で行なわれている取引(駆け引き、売買)
せ あいとい
での競りとか 相対とか正価とかに注目するのではなく、消費者の需要と生産者の供給という変数から抽象的・一般的に捉えたものです。

また、複雑な相互作用が行なわれている場をひとつのシステム（たとえば、社会システム）というトータルな

捉え方)として把握したい場合には、それをいくつかのサブシステムに分けることもあります。いずれにせよ、現実の複雑な世の中から社会(の一側面)を取り出して「◎◎システム(体系、あるいは単に系)」として捉えるとき、◎◎と名付けた実態を、実態そのものではなくすっきりと理解し、記述したいという志向性があります。

モデルという捉え方もよく使われます。これは、ある特定のシステムに対して、図示したり、数式化したり、比喩を用いたりして、そのシステムを具体化することです。これにより、システムとして捉えられたものがいつそう明確になります。言い換えると、システムとは相互作用する要素の集まりで、相互作用の結果、要素をただ単に集めた場合と異なる状態(全体性)にその特徴が現れます。これに対して、どのような要素に注目し、どのような相互関係があるのかをはっきりさせることができるのがモデル化です。

つまり、システムの中身を具体化したものがモデルです。因みに、同じシステムに対して異なるモデルを作ることが可能ですが、注目する変数や相互作用の記述がちがうために別のモデルになることもあります。同じ変数・同じ相互作用に注目しても異なるモデルになることもあります。

社会をシステムとして把握しようとする場合、大昔(古代ギリシアの時代)から現在まで使われているモデル化の一方法が類型です。類型とは、実際の現実対象をいくつかのモデル(理念型、理想型)のどれかに対応させるシステム認識です。たとえば、古代ギリシア人は、政治システムを民主制、貴族制、僭主制などのモデルに分けて、アテネとかスパルタの政治制度がどれにあたるのか、どのモデルが望ましいのか、といった議論をしていました。

近代になると、数式によるモデル化が力学で発達し、数学の発達と物理学の発達の相乗作用が進み、やがて20世紀になると、変数の読み替えにより古典的物理数学が経済数学にも通じるようになります。社会システムを数式で表現したものを数理モデルと呼びます。特に主体の相互作用に注目する数理モデルとしてゲームがあり、20世紀の後半にゲーム理論という数理モデル自体の研究が急速に進展しました。

システムで実際に何が起こるのか(つまりシステムの性質)は、具体的にはモデルを調べることによってわかつてきます。数理モデルは方程式の形をとることが多く、その場合、方程式を解く(方程式の解を求める

る)ことでシステムの特徴が明らかになります。さて、シミュレーションもシステムの性質を探るひとつの方法で、モデルを調べることと密接に関連しています。シミュレーションとは、やや粗雑な言い方をすると、モデルを動かしてみることです。シミュレーションをしやすいモデルとしにくいモデルがありますが、決して数理モデルに限定されているわけではありません。たとえば船の性能を調べるために、模型(=モデル!)をプールの中で動かしてみるのもシミュレーションです。

シミュレーションは日本語で模擬実験と呼ばれることがあります。現実の中で操作する実験とは異なり、現実の代わりに特定のモデルを操作しているので、シミュレーションは本物ではなく「模擬」の実験だからです。しかし、本物ではないからといってばかにしてはいけません。模擬試験が本物の入学試験の代りにはならないからといって、役立たないわけではありません。それどころか、非常に役立ち得ることと似ています。

社会の分析・理解の方法として、システム・モデル・シミュレーションという3つの捉え方を簡単にまとめてみました。もともと、これらは社会の捉え方に固有なものではなく、さまざまな学問分野で用いられており、システム・モデル・シミュレーションは分野毎に特色ある意味を持っています。ここでは、システム・モデル・シミュレーションの一般的な説明は省き、もっぱら社会を取り扱う場合に限って説明してみましたが、この3つの互いに関連する現実世界の捉え方は[図38.1](#)のようにまとめることができるでしょう。

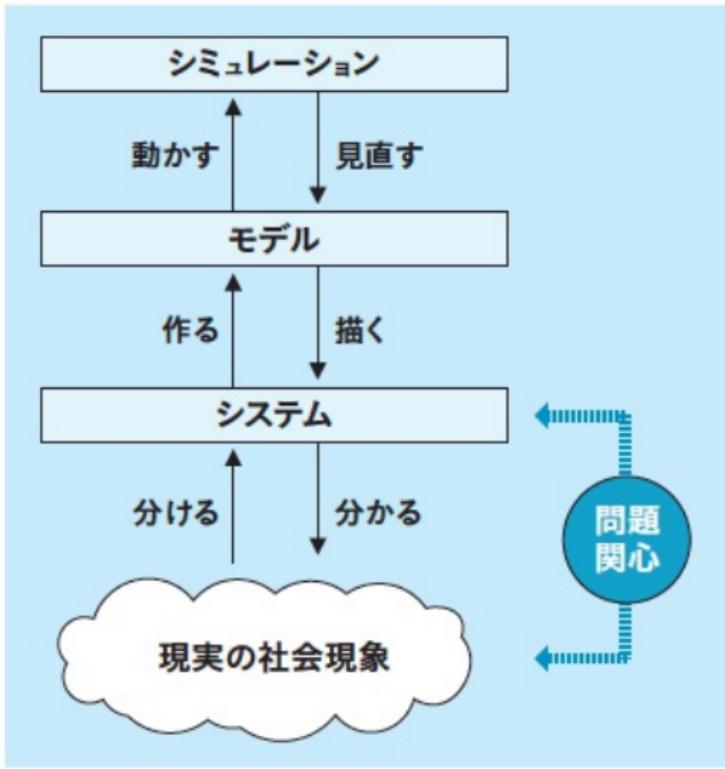


図38.1

シミュレーションという手法が特に役立つ場合がいくつかあります。モデルが数式の演算(=操作)によって(つまり解析的に)解が求まるような形式で与えられていない場合、数値を代入して「解」の存在場所の見当をつけなくてはなりませんが、この方法がシミュレーションの一形態です。方程式一般を考えると解が得られる方がむしろ例外的で、従来は、解けない方程式を解ける方程式(たとえば線形方程式)に近似させて処理してきました。その意味では、シミュレーションの方が普遍的というか汎用性があるといえるでしょう。さらに、解ける数理モデルにとっても、モデルがきわめて複雑である場合とか、時間的変化を追うことによってシステムの性質を知りたい場合には、シミュレーションが有効です。

社会システムには確率的な要素が頻繁に登場します。そのような確率的な要素を、乱数を用いて扱うことでもシミュレーションが得意とする技法です。もちろん、特定の確率分布を仮定し、期待値(平均値)とか分散を用いて、解析的に解ける数理モデルを作ることも可能です。しかし、無理にそうするよりも、あるいは数理モデルの分析と並行して、シミュレーションを行なうことは社会現象の理解に有効な方法です。

近年のシミュレーション技法の発達は、従来扱いにくかったシステムの研究をも容易にしました。「人工社会」も新しく可能になった社会の捉え方のひとつです。これは、コンピュータの中に、自分が注目するシステムが何であれ、主体どうしの相互作用や主体と環境との相互作用を中心に、ある特定のモデルとして構築し、それをシミュレーションすることによって、現実の社会についての理解を深める方法です。人工社会を作るには、数理モデルを作る必要がないどころか、システム全体を記述するモデルさえ必要ありません。必要なのは、難しい数式やプログラミングに習熟することではなく、この新しい発想に慣れることなのです。

38.4 人工社会の得意技

人工社会の視点による社会の捉え方は、従来は避けていたさまざまな問題を取り上げることを可能にしました。人工社会の技法を身につけることによって、社会科学・人間科学のフロンティアを開拓できるでしょう。しかし、人工社会は無敵の手法というわけではありません。どのような問題を扱うのに適しており、どのような使い方をすると人工社会が持っている可能性を引き出せるのか知しておくことが重要です。以下に、人工社会の手法を用いるのに特に適していると思われる問題設定を掲げます。もちろん、それらは互いに関連しています。

◎相互作用の局所性

=重要な相互作用のあり方は局所的である

第1章(1.3)で触れたように、人工社会はいわゆるボトムアップ・アプローチであり、局所現象として捉えることのできる主体間の相互作用に注目するモデル化に適しています。主体が空間的に偏在・遍在していてもかまいません。システム全体についてモデルを組み立てる必要がないので、私たちが全体としてどのような社会的相互関係になっているのか仮説を持っている必要もありません。

◎限界を持つ個

= 主体は自分の周囲(近傍)のことしかわからない

従来の社会科学では、非現実的なほど全体の状況を把握して、合理的な意思決定をする主体を前提にして、モデルを作るのがふつうです。しかし、人工社会ではそのような仮定は不要です。主体は、システム全体についての知識を持っている必要も、超人的な合理的計算能力を持っている必要もありません。

◎多様な個

= 主体の個性が重要である

従来の社会科学で登場するモデルでは、主体は没個性的ないし平均的にしか扱われない傾向があります。特に方程式によるモデル化は、変数の間の一般的な関係を定式化することをめざし、均質的な個を想定します。それに対し、人工社会では個が基本になり、それに多様な属性や関係が付随します。同じタイプの主体の間でも個性を重視します。

◎反応する個

= 相互作用や環境によって行動が変わる

人工社会で生じる相互作用は、主体どうしでも主体と環境との間でも、主体の行動に影響を与えることが可能です。そのことにより、同期、協調、伝播、流行などと呼ばれている集団的現象を的確に表すことができます。

◎変化していく個

= 相互作用の中で個は変わる

人工社会の中で変わるのは行動だけではありません。主体の属性自体も変わっていくことが可能です。特定の個が変化していく慣れや学習はもちろん、個のレベルでは変化しないものの世代交代により新しい個に変化を引き起こすことが可能です。つまり、集団として変化していく適応、進化といった現象も人工社会では生じ得ます。

◎階層的・重層的な社会

= 異なるタイプの主体の間にはレベルの違いがある

従来の社会科学では多層な主体間の相互作用を扱うことはあまり得意ではありませんでした。たとえば方法論的個人主義という考え方とは、同質・等質的な個が唯一の要素主体であり、集合的(社会的)現象をそのような主体間の相互作用だけで説明しようとするものです。個と全体との間に

あるさまざまな中間的な組織は無視されるか、要素主体間の相互作用に還元されてしまいます。人工社会では、レベルの異なる複数の主体を設定することができます。要素主体としての個をさまざまな組織や団体に結びつけることが可能です。

◎均衡よりは過程

=均衡だけではなく過程の様子を知ることが重要である

人工社会では本質的にダイナミックなシミュレーションが行なわれます。初期状態から逐次的に人工社会の状態を追いかければ、人工社会の変化を追跡することができます。遅かれ早かれ均衡点に達して静的になるかも知れませんが、いつまでもダイナミックに変動し続けるかも知れません。いいかえれば、安定的な社会システムでなくともいっこうにかまわないのです。

◎予想しにくい全体の挙動

=システム全体の様相の変化が重要である

主体間の相互作用よりも、むしろ全体に生じる変化に関心があり、特にそれが予測しにくい場合に、人工社会は力を発揮します。相転移、創発、カオス、複雑性などと呼ばれている大局的なシステムの特徴を、シミュレーションを実行しながら見ることができます。

◎中数の世界

=2者間関係にも平均的・統計的指標にも近似できない

2主体からなる社会のモデル化をわざわざ人工社会で試みるメリットはありません。反対に、大数の法則が十分成り立つような多数の個体からなる均質的な社会も、局所的相互作用のモデル化を得意とする人工社会の対象とする必要はないでしょう。少数の場合も非常に多数の場合も数理的あるいは数量的・統計的に扱う方法が発達しています。人工社会は、顔見知りの社会とか、同一主体が状況によって役割を変えるとか、主体の違いに注目したり、社会全体を表す単純なマクロ指標では表すことが難しい社会現象に注目したりする場合に適しています。

◎発見的探索

=局所的なモデルから大域的モデルを作りたい

主体間の局所的な相互作用を中心にモデル化する人工社会であっても、シミュレーションの過程で、大局的に見て顕著な特徴(何らかの秩序の形成)が生じたならば、やはりミクロとマクロの関係をきっちりと定式化したくなるでしょう。人工社会は、どのようにミクロとマクロとを結びつければよいのかについてのヒントを与えてくれるかも知れません。

自然科学では、20世紀の後半から、複雑性やカオスに関わる数理モデルやコンピュータ・シミュレーションが急速に発達しています。人類が獲得したこのような強力な方法のレパートリーの中からめぼしいものの見当をつける上で、人工社会はまず私たちが手をつけるべきひとつ的方法になり得るでしょう。

38.5 未開拓の沃野へ

私の専門はシミュレーションでも情報科学でもありません。その私が、なぜ（素人のくせに）人工社会の開発に関わったのか。それは、一言で言えば、この新しい方法に魅せられたからです。自分の携わってきた学問にとって人工社会はきわめて重要な役割を果たすと確信しました。しかし、私にはマルチエージェント・シミュレーションのプログラムを作る能力がありません。私にも使えるシミュレータが欲しいという願いがartisocの登場で実現しました。これからは、artisocを自分の専門分野で使っていく段階です。

それでは、なぜ人工社会の技法に関する本を書いたのか（余計なことをせずに自分の専門分野で応用すればよいのではないか）。それは、さまざまな分野で人工社会の技法を試してもらいたいからです。多くの分野への応用は、私には不可能です。この新しい方法に関心を持っている人に語りかけること、人工社会に対する人々の関心を高めること、がせいぜい私にできることです。

これからは読者の皆さんのがんばりの出番です。それぞれの関心に従って、人工社会の可能性を追求していく下さい。

人工社会の技法が、artisocの使いやすさも手伝って、多くの分野に浸透していくことを願っています。また、既にマルチエージェント・シミュレーションが市民権を得ている理工系の分野でも、artisocの使いやすさが学問の発展に少しでも貢献するなら望外の喜びです。

読書案内——高みをめざして

人工社会をもっと本格的に勉強したい人への、そしてartisocをさまざまな分野で使ってみたい人への、

1. 日本語で読めるものを中心に、2. 最初の手がかりになる、3. 比較的に手に入りやすく、4. しかも参考になりそうな本についての、独断と偏見に満ちた読書案内です。ここで取り上げた本の多くに、参考にすべき文献やウェブサイトがさまざまな観点から紹介されているので、「芋蔓」式に、高度な文献に挑戦できます。

◎高みをめざす前に、据野を広くしておこう(あるいは単に気分転換に)

- イリヤ・プリゴジン、イザベル・スタンジェール『混沌からの秩序』(みすず書房、1987年)
- ジェイムズ・グリック『カオスー新しい科学をつくる』(新潮社、1991年)
- スティーブン・レビー『人工生命ーデジタル生命の創造者たち』(朝日新聞社、1996年)
- M・ミッケル・ワールドロップ『複雑系』(新潮社、1996年)
- 安田雪『ネットワーク分析ー何が行為を決定するか』(新曜社、1997年)
- ポール・オームロッド『バタフライ・エコノミクス ー 複雑系で読み解く社会と経済の動き』(早川書房、2001年)
- アルバート=ラズロ・バラバシ『新ネットワーク思考 ー 世界のしくみを読み解く』(NHK出版、2002年)
- ロバート・アクセルロッド、マイケル=コーベン『複雑系組織論』(ダイヤモンド社、2003年)
- ダンカン・ワット『スマールワールド・ネットワーク ー 世界を知るための新科学的思考法』(阪急コミュニケーションズ、2004年)
- スティーヴン・ストロガツ『SYNC(シンク) ー なぜ自然はシンクロしたがるのか』(早川書房、2005年)
- マーク・ブキャナン『複雑な世界、単純な法則ーネットワーク科学の最前線』(草思社、2005年)

以上の10冊余の本はいずれも読み物だが、モデルにしたい例がたくさん見つかる。この本で作ったモデルも、いくつか紹介されている（分居モデル、ライフゲーム、ボイド・モデル、ランダム・ネットワーク、DLAなど）。日本語原著が1冊だけなのがなんとなく癪に障る。年代順に配列してみたが、20年（一世代）で生じた世界認識の変化は実に印象的だ。

- ナイジェル・ギルバート、クラウス・G・トロイチュ『社会シミュレーションの技法 — 政治・経済・社会をめぐる思考技術のフロンティア』（日本評論社、2003年）
 - さまざまなタイプ（人工社会もその一つ）のシミュレーションを概説。「入り口」しかないが、各入り口から入った後にどのように詳しく学べばよいか示した道案内がある。第1章と第2章の一般論も有益。
- Mitchel Resnick『非集中システム』（コロナ社、2001年）
 - シミュレータStarLogoを用いたマルチエージェント・シミュレーションの紹介。事例が豊富で示唆的。サンプルの分居モデル（第2章で紹介）で、なぜヒトではなく「カメ」がエージェントなのかという素朴な疑問も、本書を読めば氷解するはず。

◎人工社会についてもっと知ろう

- Thomas C. Schelling, *Micromotives and Macrobbehavior*, W. W. Norton & Company, 1978
 - 人間社会におけるミクロレベルとマクロレベルとの関連・乖離を扱った（おそらく）嚆矢。さまざまなアイデアの宝庫。翻訳がないのが不可解。ここに所収されている分居モデル（第10章で作成）は、マルチエージェント・シミュレーションや複雑系に関する本では必ずといってよいほど紹介されている。
- ロバート・アクセルロッド『つきあい方の科学 — バクテリアから国際関係まで』（ミネルヴァ書房、1998年）
 - さまざまな分野に影響をおよぼしたマルチエージェント・シミュレーションの古典。書名をなぜ「協力行動の進化」と原題の直訳にしなかったのだろうか。ゲーム戦略選手権モデル（第29章で作成）や戦略の適応モデル（第30章で作成）の元版を紹介。

- Joshua M. Epstein and Robert Axtell『人工社会－複雑系とマルチエージェント・シミュレーション』(共立出版, 1999年)
 - ひとつの基本的なモデル(「アリ」が主人公)に、流行や戦争などのさまざまな社会現象を結びつける巧みな発想と解釈が光っている。原書のタイトルで「人工社会」が初めて登場した(たぶん)。
- 山影進、服部正太編『コンピュータのなかの人工社会－マルチエージェントシミュレーションモデルと複雑系』(共立出版, 2002年)
 - artisocの前身であるKK-MASを用いたモデル実例集。抽象的なものから具体的なものまで多種多様なモデルが紹介されている。

◎複雑な人工社会の構築に向けて

- 生天目章『マルチエージェントと複雑系』(森北出版, 1998年)
 - エージェントの相互行為についての基本書。artisocで作ってみたいモデルが満載。数式が登場するものの、ルール化はそんなにむずかしくない。
- 山田誠二『適応エージェント』(共立出版, 1997年)
 - エージェントの学習(環境との相互作用の改善)を定式化。技法よりは考え方の解説。
- ロバート・アクセルロッド『対立と協調の科学－エージェント・ベース・モデルによる複雑系の解明』(ダイヤモンド社, 2003年)
 - 社会秩序変動に関するさまざまな話題とそのモデル化。文化変容モデル(第23章で作成)の元版を所収。全体の統一性はないが、使えそうなアイデアが散見。
- 大内東、山本雅人、川村秀憲『マルチエージェントシステムの基礎と応用－複雑系工学の計算パラダイム』(コロナ社, 2002年)
 - ゲーム、学習、遺伝的アルゴリズムなど盛りだくさんの内容だが、「必要最低限の内容」(まえがき)だそうだ。たしかに本書の内容をマスターすれば、さまざまな問題に取り組める。artisocのモデル化に挑戦するときは、参考文献にも目を通そう。

- 生天目章『ゲーム理論と進化ダイナミクス－人間関係に潜む複雑系』(森北出版, 2004年)
 - － 標準的なゲーム理論から進化ゲーム理論までカバーして相互作用を類型化。ゲーム理論に基づきかけられた人工社会の構築をめざす人向け。
- 金光淳『社会ネットワーク分析の基礎－社会的関係資本論にむけて』(勁草書房, 2003年)
 - － 「基礎」とは「入門」ではなく「土台」。応用範囲は広く、ネットワーク理論に基づきかけられた人工社会の構築をめざす人向け。

◎さまざま分野でartisocを使ってみよう

- 増田直紀, 今野紀雄『「複雑ネットワーク」とは何か－複雑な関係を読み解く新しいアプローチ』(講談社, 2006年)
 - － ネットワーク理論の入門書。ポイントを簡潔に押さえてあるので、さまざまなタイプのネットワークをartisocでモデル化できる。
- 増田直紀, 今野紀雄『複雑ネットワークの科学』(産業図書, 2005年)
 - － 豊富な内容をコンパクトに解説。artisocでグラフ理論やその応用(ネットワーク)をモデル化する際に数学的基礎が欲しい人は必読。上記『「複雑ネットワーク」とは何か』が物足りなくなった人向け。
- 北中英明『複雑系マーケティング入門－マルチエージェント・シミュレーションによるマーケティング』(共立出版, 2005年)
 - － 「マーケティングの領域における意思決定の際の新しいツールとしてマルチエージェント・シミュレーションの適用を試みたもの」(あとがき)だが、一般的な議論も充実している。マーケティングに関心ない人も読む価値あり。シミュレーションにはKK-MASを使用。当然、artisocによるモデル化が可能。
- 和泉潔『人工市場－市場分析の複雑系アプローチ』(森北出版, 2003年)
 - － 人工市場の基礎、作り方、使い方からなる3部構成の、簡にして要を得た解説。artisocで人工市場モデルを作つてみることも可能。

- ポール・クルーグマン『自己組織化の経済学 — 経済秩序はいかに創発するか』(東洋経済新報社, 1997年)
 - エッジシティと呼ばれるリングワールド(といってもSFではない)でのお話。国際経済を立地論と結びつけた同一著者による『脱『国境』の経済学』(東京経済新報社, 1994)と併せて。
- W. ブライアン・アーサー『収益遞増と経路依存 — 複雑系の経済学』(多賀出版, 2003年)
 - 複雑系経済学のテーマをほぼ網羅。数理モデルによる説明のため、やや難解かも知れないが、artisocでモデルを作り、数値的に納得してみては。
- 友野典男『行動経済学 — 経済は「感情」で動いている』(光文社, 2006年)
 - 人間被験者を使った多数の実験に基づいた新しい経済人像の提示。人間の頭の中はブラックボックス(今のところ)だが、人工社会のエージェントを使えばホワイトボックス化が可能。
- 高玉圭樹『マルチエージェント学習 — 相互作用の謎に迫る』(コロナ社, 2003年)
 - エージェントの学習について工学の立場から詳しく紹介。マルチエージェントシステムそのものについての解説も明快。学習エージェントを作りたい人は必読。エージェントどうしの協力を学習させることの難しさも伝わってくる。
- 香取眞理『複雑系を解く確率モデル — こんな秩序が自然を操る』(講談社, 1997年)
 - 物理学から疫学まで、いろいろなタイプの格子型のモデルを紹介。artisocでも簡単にモデル化が可能。臨界値や相転移についても理解できる。流行モデル(第13章で作成)も格子(セル)型モデルとして登場。
- 宮下精二『相転移・臨界現象 — ミクロなゆらぎとマクロの確実性』(岩波書店, 2002年)
 - 相転移や臨界現象など、膨大な数の分子や原子、電子の織り成す不思議で複雑な世界に関心がある人はこちら。格子ガスモデル(同書では「モデル」を「模型」と言っている)やハイゼンベルクモデル、イジングモデルなどの古典的な統計力学のモデルとシミュレーション例がコンパクトに紹介されている。

- 都甲潔, 江崎秀, 林健司『自己組織化とは何か－生物の形やリズムが生まれる原理を探る』(講談社, 1999年)
 - － 人工生命も扱われているが、粘菌の不思議なふるまいから脳の形成、量子レベルのマイクロマシンから人工の細胞まで、コンピュータの外で人工物と自然—その境界も曖昧であることが実感できる—が展開するさまざまな自己組織化の紹介が中心。artisocを使ってコンピュータの中にフィードバックさせてみるのもいいかもしれない。
- カール・シグムンド『数学でみた生命と進化－生き残りゲームの勝者たち』(講談社, 1996年)
 - － 数学(数式)は登場しないので心配無用。原題は『生命をめぐるゲーム』。ライフゲーム(第20章で作成), ゲーム戦略選手権モデル(第29章で作成), 適応モデル(第30章で作成)も紹介されている本書は、生物学のマルチエージェント・シミュレーションの入門書。説明は具体的なので、artisocによるモデル化も比較的容易。
- 土場学ほか編『社会を〈モデル〉でみる－数理社会学への招待』(勁草書房, 2004年)
 - － 個人から全体社会までさまざまな問題からなる44の「なぜ」(もちろん分居も含む)をモデルで説明。社会現象をartisocでモデル化する際のサンプル集になる。
- 亀田達也, 村田光二『複雑さに挑む社会心理学　－　適応エージェントとしての人間』(有斐閣, 2000年)
 - － ミクロ・マクロレベルの相互作用を前面に社会心理学のさまざまな知見を紹介している。社会的影響過程を扱ったシミュレーションモデルや合議のモデルなども扱われており、さまざまな着想を得ることができる。

◎土台をしっかりと

- 廣瀬通孝, 小木哲朗, 田村善昭『シミュレーションの思想』(東京大学出版, 2002年)
 - － 「シミュレーションを実施するにあたっての基本的考え方についての話題が中心」(あとがき)の本書は、工学系の発想だが、文系の人間にも示唆するところが多い。

- 松野孝一郎『内部観測とは何か』(青土社, 2000年)
 - 人工社会の形而上学的基礎を考えたい人向け。モデルの作り方(エージェントの相互作用の記述)だけでなく、さまざまな設定(初期値, 実行環境, 出力, その他)についても深く考えさせられる。安直なポストモダンと同列に置いてはいけない。
- 松原望『入門確率過程』(東京図書, 2003年)
 - 亂数(確率)を多用する人工社会の技法に数学的根拠づけを必要とする人向け。人工市場(金融工学)の理論的基礎まで(途中で落ちこぼれなければ)連れて行ってくれる。artisocのモデル化では, Rnd()の魔術師になれるかも。
- 小林健一郎『これならわかるC++ – 挫折しないプログラミング入門』(講談社, 2001年)
 - artisocのルールで中括弧(たとえばAgt_Init{ })がなぜ使われるのか, 括弧内がカラの関数(たとえばRnd())がなぜあるのか, といったことが気になって夜も眠れない人のために。人工社会が依拠しているプログラミング文化の一端に触れることができる。

◎汎用マルチエージェント・シミュレータの例

- RePast URL: <http://repast.sourceforge.net/>
- Mason URL: <http://cs.gmu.edu/~eclab/projects/mason/>
- Soars URL: <http://www.cs.dis.titech.ac.jp/ja/>
- PlatBoxURL: <http://platbox.sfc.keio.ac.jp/>
- U-mart URL: <http://www.u-mart.org/html/index-j.html>

これらは現在、社会科学分野でプログラミングに長けた研究者たちが競って開発しているシミュレータについてのウェブサイトである。覗いてみると、artisocの特徴がわかるに違いない。

◎蛇足

- 構造計画研究所のMASコミュニティ <http://mas.kke.co.jp/index.php>

- 山影研究室MASページ <http://citrus.c.u-tokyo.ac.jp/mas/index.htm>

artisoc(やその前身のKK-MAS)によるモデルやそれらに基づいた研究については、上のサイトを見るに
かぎる。

作成モデルの概要

飛ぶ鳥(3, 4章)

- 多数のエージェントが空間の中心から様々な方向に向かって真っ直ぐ飛びます。まるで波紋のように広がっていきます。
- 単純なモデルを作る事で, artisocでのモデル作りの流れを理解してもらいます。また, エージェントのルールとして空間上を移動する方法の基本を学びます。

花火(5章)

- 多数のエージェントが一丸となって空間の下の端から上に移動し, 30ステップ後に様々な方向に向かって飛び散ります。
- 条件を指定して行動の場合分けをする技法の基礎を学びます。

立ち話(6章)

- 周りに人やペットがいたら立ち止まります。
- 周囲のエージェントを認識して数える技法を学びます。また, その数に応じて行動を変化させる方法も学びます。

立ち話2(7章)

- 立ち話モデルの人数や視野をコントロールパネルから設定できるようにしたモデルです。
- コントロールパネルの設定方法, Universeのルール内でのエージェントの生成, 配置方法の基礎を学びます。

立ち話3(8章)

- 立ち止まっている人の数や割合を時系列グラフに出力するようにします。全員が立ち止まると, シミュレーションは自動的に終了します。
- エージェントの集計, 時系列グラフの設定, シミュレーションをルールで自動終了させる技法を学びます。

空き地(9章)

- ・ 罫線の描かれた空間を、エージェントが移動します。
- ・ artisocにおける二次元空間の成り立ちを理解してもらうためのモデルです。

映画館(9章)

- ・ 近くの席が混んでいると空いているところに移動します。
- ・ 格子上空間での座標、移動方法を学びます。

分居モデル(10章)

- ・ 同じ種類の仲間が一定数以上居ないと引っ越してしまいます。それほど排他的でなくとも、分居が進んでしまいます。
- ・ 第1部で学んだ技法を用いて、有名な「分居モデル」を作ってもらいます。ノーベル賞受賞者の研究を再現できたという達成感を味わって下さい。

ラッシュアワー(11章)

- ・ 駅のコンコースで東西に行き交う人々をモデル化します。人の流れが出来てきます。
- ・ 自分の周囲の特定の方向の様子を調べる技法とともに、複雑な場合分けを学びます。

国の規模(12章)

- ・ 二次元空間をグラフに見立てて、仮想国家の人口と経済の成長を表示します。
- ・ 二次元空間の変わった使い方をしながら、頭を柔らかくしてもらいます。また、エージェントの状態によって色を変化させる技法を学びます。

風邪の流行(13章)

- ・ 周囲に風邪を引いている人がいると、一定の確率で風邪をうつされます。風邪はどんなふうに流行するのでしょうか？
- ・ 周囲にいる全てのエージェントの状態を調べる技法、そして調べた結果によって影響を受ける技法を学びます。

蝶のつがい探し(14章)

- ・ オスがメスを探して飛び回ります。
- ・ 周囲のエージェントの中から、ランダムに一体だけを選び出して、相手の状態を確認する技法を学びます。

牧羊犬(15章)

- 牧羊犬が羊を柵の中に集めます。
- 他のエージェントの状態や行動を変化させる技法を学びます。

生態系(16章)

- 動物プランクトンが植物プランクトンを食べます。捕食者と被捕食者の個体数変動を見てみましょう。
- エージェントに、他のエージェントを生成させたり、削除させたりする技法を学びます。

幼稚園(17章)

- 先生が幼稚園の子供達を東に歩かせようとします。が、子供達は言われた事なんてすぐに忘れてしまいます。
- 自律的なエージェントに対して、他者が影響を与える例です。東を向くのでも、先生が強制的に向かせるのと、他の子供達が東を向いていたら自分も向くのでは、中身も結果も違います。

暴走族の取締(17章)

- 警官がスピード違反した運転手に切符を切れます。何度目の違反かによって、処分の重さが違います。
- これも自律的なエージェントに対して、他者が影響を与えるモデルです。これは段階的に相手の行動に制限を加える例です。

深海魚(18章)

- 深海魚が、海の深いところだけを泳ぎ回ります。
- 海の深さを空間変数で表現します。エージェントが空間変数を見て行動を変えるようにする技法も学びます。

空気感染(18章)

- 病気の人がいた空間に病原体が残り、それが原因で感染が広がります。
- 空間変数で表したもの(病原体)が、時間とともに変化する(減ってゆく)技法を学びます。

森林火災(19章)

- 森林火災の広がり方の違いは意外なところに原因があるかも。
- 同期問題とは何かを理解し、その解決方法を学んでもらいます。また、繰り返し文を途中で中止する技法も学びます。

ライフゲーム(20章)

- 生き物と同様、過疎でも過密でも死んでしまうセルが並んでいます。様々なパターンが生ま
では消えてゆきます。
- 第2部で学んだ技法を用いて、有名な「ライフゲーム」を作ってもらいます。

狼と羊(21章)

- 狼が羊を追いかけます。しかし、ただ追いかけるといっても、その中には様々な行動要素が入っ
ています。
- 空間の端に来たら方向を変えたり、目標の方向を向いたり、近くに来たら速度を落としたりと、
様々な動き方の技法を学びます。

パーティ(22章)

- 立食パーティで、興味を持った相手と話をします。
- 性別などの属性を、文字列型変数で表します。文字列型変数の操作方法や、色との関連
付け方法を学びます。

文化変容(23章)

- 文化的に多様な共同体がいます。文化は周囲の共同体の文化に影響されて変わっていきま
す。類似した文化からはより強く影響を受けます。
- 文字列の操作に慣れてもらいます。文字列どうしを比べたり、マップ上に存在する文化の種類
を数える技法も紹介します。

友達の輪(24章)

- マンガと音楽の趣味が近い人と友達になります。どのような友達のネットワークができるでしょう
か?
- 複数のエージェント集合をくっつけたり、共通部分を抜き出したり、引き算したりする技法を学び
ます。また、空間上でエージェントどうしを線で結ぶ方法も紹介します。

クラスメイト(25章)

- 同級生と遊んだりけんかしたりします。するとその関係が蓄積されて、好き嫌いが決定されま
す。
- エージェントをIDを用いて個体識別する技法、それぞれとの関係を配列のある変数に格納す

る技法を学んでもらいます。

ボイド(26章)

- 鳥が群れを作つて飛ぶ様子をモデル化した「ボイドモデル」の簡易版を作ります。
- SubとFunctionという2種類のユーザ定義関数を用いて、ルールをすっきりと記述する方法を紹介します。

ランダム・ネットワーク(27章)

- エージェントが、ランダムな相手と徐々に繋がっていきます。ネットワークはどのように出来てゆくのでしょうか？
- 個体識別、配列、エージェントの円形配置、最大値や最小値の見つけ方など技巧的な方法をいくつも組み合わせる事に慣れてもらいます。

世界帝国の形成(28章)

- 国々が戦い、相手の領土を奪い合います。最終的には一つの帝国ができあがります。
- 六角形のマス目を使ったモデルの例です。四角形のマスの時との視覚的な違いを実感してもらいます。

会社で出世する方法(28章)

- ヒラ、管理職、役員の三階層からなる会社があります。新入社員がだんだん出世していく様子を観察しましょう。
- レイヤを使った階層的なモデルの例です。表示方法や利用方法ばかりでなく、レイヤが違うということの意味も考えてみてください。

囚人のジレンマ(29、30章)

- エージェント同士が「囚人のジレンマ」ゲームで繰り返し対戦するモデルです。30章ではエージェントが強い他者の戦略を真似て「学習」します。
- 過去の情報に基づいて行動する技法を学びます。また、学習や自然淘汰によって、優秀な戦略が広がっていく方法を紹介します。

追跡(31章)

- 狹い路地を逃げる泥棒を、警官が追いかけます。
- ステップ内におけるエージェントの実行順序の違いが結果に影響する事を理解して貰うためのモデルです。

世界地図(33章)

- 背景に世界地図を表示し、主要国の首都をエージェントとして適切な場所に配置します。
- 二次元空間の背景に画像を表示する方法、および、エージェントの初期値をファイルから読み込む方法を実演するためのモデルです。

零戦(33章)

- 零戦が空で旋回します。
- エージェントを画像で表示する方法、背景画像やエージェント画像をルールで切り替える方法を実演するためのモデルです。

雪国(34章)

- 地面に雪が降り積もり、とけてゆきます。
- 空間変数しか用いないで、状態の変化を表現するモデルです。

航海(34章)

- 船が航跡を残しつつ進んでゆきます。
- 空間変数で表したものが、時間とともに変化する技法を学びます。

拡散律速凝集(34章)

- 飛び交っている粒子が別の粒子にぶつかるとくっついて止まります。粒子の塊はどのような形に育ってゆくでしょうか？
- 空間変数を用いてエージェントの数を節約する技法を実演するモデルです。エージェントの数が増えると、実行するルールが増えてモデルが遅くなってしまいます。

富士登山(34章)

- 標高データを読み込んで富士山を再現し、エージェントに登らせます。尾根登りと沢登りがあります。
- 空間変数のファイルからの読み込みを実演するモデルです。周囲の空間変数を調べる技法も学びます。

著者・執筆協力者

著者：山影進（やまかげ すすむ）

- 1972年東京大学卒業、1982年マサチューセッツ工科大学Ph.D.、1991年より東京大学教授。
- 専門は国際関係論、特に国際関係理論、比較地域体系、ASEAN。
- 最近の著作は『国際関係論』（小和田恒と共著、2002年）、『コンピュータのなかの人工社会』（服部正太と共に編著、2002年）、『東アジア地域主義と日本外交』（編著、2003年）など。

執筆協力者（50音順）

- 阪本拓人（東京大学リサーチフェロー）
- 鈴木一敏（東京大学リサーチフェロー）
- 保城広至（東京大学助手）
- 光辻克馬（東京大学リサーチフェロー）
- 山本和也（東京大学リサーチフェロー）