

# Think Complexity

Version 2.6.3

思考复杂性

2.6.3 版本





# Think Complexity

Version 2.6.3

Allen B. Downey

Green Tea Press

Needham, Massachusetts

思考复杂性

2.6.3 版本

艾伦·b·唐尼

绿茶出版社

马萨诸塞州，尼德姆

Copyright © 2016 Allen B. Downey.

Green Tea Press  
9 Washburn Ave  
Needham MA 02492

Permission is granted to copy, distribute, transmit and adapt this work under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License: <http://thinkcomplex.com/license>.

If you are interested in distributing a commercial version of this work, please contact the author.

The L<sup>A</sup>T<sub>E</sub>X source for this book is available from

<https://github.com/AllenDowney/ThinkComplexity2>

版权所有 2016 艾伦 ·b·唐尼。

绿茶出版社  
9 沃什伯恩大街  
02492

许可授予复制，分发，传输和改编此作品的知识共享属性-非商业性-共享相同方式  
4.0 国际许可证: <http://thinkcomplex.com/License>。

如果你有兴趣发行这部作品的商业版本，请与作者联系。

这本书的 `l a t e x` 来源可从

<https://github.com/allendowney/thinkcomplexity2>





# Contents

<b>Preface</b>	<b>xi</b>
0.1 Who is this book for? . . . . .	xii
0.2 Changes from the first edition . . . . .	xiii
0.3 Using the code . . . . .	xiii
<b>1 Complexity Science</b>	<b>1</b>
1.1 The changing criteria of science . . . . .	3
1.2 The axes of scientific models . . . . .	4
1.3 Different models for different purposes . . . . .	6
1.4 Complexity engineering . . . . .	7
1.5 Complexity thinking . . . . .	8
<b>2 Graphs</b>	<b>11</b>
2.1 What is a graph? . . . . .	11
2.2 NetworkX . . . . .	13
2.3 Random graphs . . . . .	16
2.4 Generating graphs . . . . .	17
2.5 Connected graphs . . . . .	18
2.6 Generating ER graphs . . . . .	20
2.7 Probability of connectivity . . . . .	22

## 内容

Preface	xi
0.1 Who is this book for? . . . . .	xii
0.2 Changes from the first edition . . . . .	xiii
0.3 Using the code . . . . .	xiii
1 Complexity Science	1
1.1 The changing criteria of science . . . . .	3
1.2 The axes of scientific models . . . . .	4
1.3 Different models for different purposes . . . . .	6
1.4 Complexity engineering . . . . .	7
1.5 Complexity thinking . . . . .	8
2 Graphs	11
2.1 What is a graph? . . . . .	11
2.2 NetworkX . . . . .	13
2.3 Random graphs . . . . .	16
2.4 Generating graphs . . . . .	17
2.5 Connected graphs . . . . .	18
2.6 Generating ER graphs . . . . .	20
2.7 Probability of connectivity . . . . .	22

---

2.8	Analysis of graph algorithms . . . . .	24
2.9	Exercises . . . . .	25
<b>3</b>	<b>Small World Graphs</b>	<b>27</b>
3.1	Stanley Milgram . . . . .	27
3.2	Watts and Strogatz . . . . .	28
3.3	Ring lattice . . . . .	30
3.4	WS graphs . . . . .	32
3.5	Clustering . . . . .	33
3.6	Shortest path lengths . . . . .	35
3.7	The WS experiment . . . . .	36
3.8	What kind of explanation is <i>that</i> ? . . . . .	38
3.9	Breadth-First Search . . . . .	39
3.10	Dijkstra's algorithm . . . . .	41
3.11	Exercises . . . . .	43
<b>4</b>	<b>Scale-free networks</b>	<b>47</b>
4.1	Social network data . . . . .	47
4.2	WS Model . . . . .	50
4.3	Degree . . . . .	51
4.4	Heavy-tailed distributions . . . . .	53
4.5	Barabási-Albert model . . . . .	55
4.6	Generating BA graphs . . . . .	57
4.7	Cumulative distributions . . . . .	59
4.8	Explanatory models . . . . .	62
4.9	Exercises . . . . .	63

---

2.8	Analysis of graph algorithms . . .	24
2.9	Exercises . . . . .	25
3	Small World Graphs	27
3.1	Stanley Milgram . . . . .	27
3.2	Watts and Strogatz . . . . .	28
3.3	Ring lattice . . . . .	30
3.4	WS graphs . . . . .	32
3.5	Clustering . . . . .	33
3.6	Shortest path lengths . . . . .	35
3.7	The WS experiment . . . . .	36
3.8	What kind of explanation is that?	38
3.9	Breadth-First Search . . . . .	39
3.10	Dijkstra's algorithm . . . . .	41
3.11	Exercises . . . . .	43
4	Scale-free networks	47
4.1	Social network data . . . . .	47
4.2	WS Model . . . . .	50
4.3	Degree . . . . .	51
4.4	Heavy-tailed distributions . . . . .	53
4.5	Barabasi-Albert model . . . . .	55
4.6	Generating BA graphs . . . . .	57
4.7	Cumulative distributions . . . . .	59
4.8	Explanatory models . . . . .	62
4.9	Exercises . . . . .	63

---

<b>5</b>	<b>Cellular Automaton</b>	<b>67</b>
5.1	A simple CA . . . . .	67
5.2	Wolfram's experiment . . . . .	68
5.3	Classifying CAs . . . . .	69
5.4	Randomness . . . . .	71
5.5	Determinism . . . . .	72
5.6	Spaceships . . . . .	73
5.7	Universality . . . . .	76
5.8	Falsifiability . . . . .	77
5.9	What is this a model of? . . . . .	78
5.10	Implementing CAs . . . . .	80
5.11	Cross-correlation . . . . .	82
5.12	CA tables . . . . .	84
5.13	Exercises . . . . .	85
<b>6</b>	<b>Game of Life</b>	<b>89</b>
6.1	Conway's GoL . . . . .	89
6.2	Life patterns . . . . .	92
6.3	Conway's conjecture . . . . .	92
6.4	Realism . . . . .	94
6.5	Instrumentalism . . . . .	95
6.6	Implementing Life . . . . .	97
6.7	Exercises . . . . .	99
<b>7</b>	<b>Physical modeling</b>	<b>103</b>
7.1	Diffusion . . . . .	103
7.2	Reaction-diffusion . . . . .	105

---

5	Cellular Automata	67
5.1	A simple CA . . . . .	67
5.2	Wolfram's experiment . . . . .	68
5.3	Classifying CAs . . . . .	69
5.4	Randomness . . . . .	71
5.5	Determinism . . . . .	72
5.6	Spaceships . . . . .	73
5.7	Universality . . . . .	76
5.8	Falsifiability . . . . .	77
5.9	What is this a model of? . . . . .	78
5.10	Implementing CAs . . . . .	80
5.11	Cross-correlation . . . . .	82
5.12	CA tables . . . . .	84
5.13	Exercises . . . . .	85
6	Game of Life	89
6.1	Conway's GoL . . . . .	89
6.2	Life patterns . . . . .	92
6.3	Conway's conjecture . . . . .	92
6.4	Realism . . . . .	94
6.5	Instrumentalism . . . . .	95
6.6	Implementing Life . . . . .	97
6.7	Exercises . . . . .	99
7	Physical modeling	103
7.1	Diffusion . . . . .	103
7.2	Reaction-diffusion . . . . .	105

---

7.3	Percolation . . . . .	109
7.4	Phase change . . . . .	110
7.5	Fractals . . . . .	113
7.6	Fractals and Percolation Models . . . . .	115
7.7	Exercises . . . . .	116
<b>8</b>	<b>Self-organized criticality</b>	<b>119</b>
8.1	Critical Systems . . . . .	119
8.2	Sand Piles . . . . .	120
8.3	Implementing the Sand Pile . . . . .	121
8.4	Heavy-tailed distributions . . . . .	125
8.5	Fractals . . . . .	127
8.6	Pink noise . . . . .	131
8.7	The sound of sand . . . . .	132
8.8	Reductionism and Holism . . . . .	134
8.9	SOC, causation, and prediction . . . . .	137
8.10	Exercises . . . . .	138
<b>9</b>	<b>Agent-based models</b>	<b>141</b>
9.1	Schelling's Model . . . . .	142
9.2	Implementation of Schelling's model . . . . .	143
9.3	Segregation . . . . .	145
9.4	Sugarscape . . . . .	147
9.5	Wealth inequality . . . . .	150
9.6	Implementing Sugarscape . . . . .	151
9.7	Migration and Wave Behavior . . . . .	154
9.8	Emergence . . . . .	155

---

7.3	Percolation .....	109
7.4	Phase change .....	110
7.5	Fractals .....	113
7.6	Fractals and Percolation Models .....	115
7.7	Exercises .....	116
8	Self-organized criticality .....	119
8.1	Critical Systems .....	119
8.2	Sand Piles .....	120
8.3	Implementing the Sand Pile .....	121
8.4	Heavy-tailed distributions .....	125
8.5	Fractals .....	127
8.6	Pink noise .....	131
8.7	The sound of sand .....	132
8.8	Reductionism and Holism .....	134
8.9	SOC, causation, and prediction .....	137
8.10	Exercises .....	138
9	Agent-based models .....	141
9.1	Schelling's Model .....	142
9.2	Implementation of Schelling's model .....	143
9.3	Segregation .....	145
9.4	Sugarscape .....	147
9.5	Wealth inequality .....	150
9.6	Implementing Sugarscape .....	151
9.7	Migration and Wave Behavior .....	154
9.8	Emergence .....	155

---

9.9	Exercises . . . . .	157
<b>10</b>	<b>Herds, Flocks, and Traffic Jams</b>	<b>159</b>
10.1	Traffic jams . . . . .	159
10.2	Random perturbation . . . . .	163
10.3	Boids . . . . .	164
10.4	The Boid algorithm . . . . .	165
10.5	Arbitration . . . . .	168
10.6	Emergence and free will . . . . .	169
10.7	Exercises . . . . .	171
<b>11</b>	<b>Evolution</b>	<b>173</b>
11.1	Simulating evolution . . . . .	174
11.2	Fitness landscape . . . . .	175
11.3	Agents . . . . .	176
11.4	Simulation . . . . .	177
11.5	No differentiation . . . . .	178
11.6	Evidence of evolution . . . . .	179
11.7	Differential survival . . . . .	182
11.8	Mutation . . . . .	183
11.9	Speciation . . . . .	186
11.10	Summary . . . . .	189
11.11	Exercises . . . . .	190
<b>12</b>	<b>Evolution of cooperation</b>	<b>191</b>
12.1	Prisoner's Dilemma . . . . .	192
12.2	The problem of nice . . . . .	193

---

9.9	Exercises . . . . .	157
10	Herds, Flocks, and Traffic Jams	159
10.1	Traffic jams . . . . .	159
10.2	Random perturbation . . . . .	163
10.3	Boids . . . . .	164
10.4	The Boid algorithm . . . . .	165
10.5	Arbitration . . . . .	168
10.6	Emergence and free will . . . . .	169
10.7	Exercises . . . . .	171
11	Evolution	173
11.1	Simulating evolution . . . . .	174
11.2	Fitness landscape . . . . .	175
11.3	Agents . . . . .	176
11.4	Simulation . . . . .	177
11.5	No differentiation . . . . .	178
11.6	Evidence of evolution . . . . .	179
11.7	Differential survival . . . . .	182
11.8	Mutation . . . . .	183
11.9	Speciation . . . . .	186
11.10	Summary . . . . .	189
11.11	Exercises . . . . .	190
12	Evolution of cooperation	191
12.1	Prisoner's Dilemma . . . . .	192
12.2	The problem of nice . . . . .	193

12.3	Prisoner's dilemma tournaments . . . . .	195
12.4	Simulating evolution of cooperation . . . . .	196
12.5	The Tournament . . . . .	198
12.6	The Simulation . . . . .	200
12.7	Results . . . . .	202
12.8	Conclusions . . . . .	205
12.9	Exercises . . . . .	207
<b>A</b>	<b>Reading list</b>	<b>209</b>

---

12.3 Prisoner's dilemma tournaments . .	.....	195
12.4 Simulating evolution of cooperation	.....	196
12.5 The Tournament . . . . .	.....	198
12.6 The Simulation . . . . .	.....	200
12.7 Results . . . . .	.....	202
12.8 Conclusions . . . . .	.....	205
12.9 Exercises . . . . .	.....	207
A Reading list		209

# Preface

Complexity science is an interdisciplinary field — at the intersection of mathematics, computer science and natural science — that focuses on **complex systems**, which are systems with many interacting components.

One of the core tools of complexity science is discrete models, including networks and graphs, cellular automata, and agent-based simulations. These tools are useful in the natural and social sciences, and sometimes in arts and humanities.

For an overview of complexity science, see <http://thinkcomplex.com/complex>.

Why should you learn about complexity science? Here are a few reasons:

- Complexity science is useful, especially for explaining why natural and social systems behave the way they do. Since Newton, math-based physics has focused on systems with small numbers of components and simple interactions. These models are effective for some applications, like celestial mechanics, and less useful for others, like economics. Complexity science provides a diverse and adaptable modeling toolkit.
- Many of the central results of complexity science are surprising; a recurring theme of this book is that simple models can produce complicated behavior, with the corollary that we can sometimes explain complicated behavior in the real world using simple models.
- As I explain in Chapter 1, complexity science is at the center of a slow shift in the practice of science and a change in what we consider science to be.

## 前言

复杂性科学是数学、计算机科学和自然科学交叉的跨学科领域，专注于复杂系统，这些系统包含许多相互作用的组成部分。

复杂性科学的核心工具之一是离散模型，包括网络和图形、元胞自动机和基于代理的模拟。这些工具在自然科学和社会科学中很有用，有时在艺术和人文科学中也很有用。

有关复杂性科学的概述，请参阅《<http://thinkcomplex.com/complex>》。

为什么你要学习复杂性科学? 以下是几个原因:

复杂性科学是有用的，特别是用来解释为什么自然和社会系统会如此行事。自牛顿以来，以数学为基础的物理学一直专注于具有少量组件和简单相互作用的系统。这些模型对于某些应用是有效的，比如天体力学，而对于其他应用，比如经济学，就没那么有用了。Complexity science provides a diversity and adaptability modeling toolkit.

复杂性科学的许多核心结果都是令人惊讶的; 本书反复出现的一个主题是简单的模型可以产生复杂的行为，其推论有时我们可以解释复杂的行为在现实世界中使用简单的模型。

正如我在第一章中所解释的，复杂性科学处于科学实践的缓慢转变和我们所认为的科学变革的中心。

- Studying complexity science provides an opportunity to learn about diverse physical and social systems, to develop and apply programming skills, and to think about fundamental questions in the philosophy of science.

By reading this book and working on the exercises you will have a chance to explore topics and ideas you might not encounter otherwise, practice programming in Python, and learn more about data structures and algorithms.

Features of this book include:

**Technical details** Most books about complexity science are written for a popular audience. They leave out technical details, which is frustrating for people who can handle them. This book presents the code, the math, and the explanations you need to understand how the models work.

**Further reading** Throughout the book, I include pointers to further reading, including original papers (most of which are available electronically) and related articles from Wikipedia and other sources.

**Jupyter notebooks** For each chapter I provide a Jupyter notebook that includes the code from the chapter, additional examples, and animations that let you see the models in action.

**Exercises and solutions** At the end of each chapter I suggest exercises you might want to work on, with solutions.

For most of the links in this book I use URL redirection. This mechanism has the drawback of hiding the link destination, but it makes the URLs shorter and less obtrusive. Also, and more importantly, it allows me to update the links without updating the book. If you find a broken link, please let me know and I will change the redirection.

## 0.1 Who is this book for?

The examples and supporting code for this book are in Python. You should know core Python and be familiar with its object-oriented features, specifically using and defining classes.

## 第十二章第 0 章序言

---

学习复杂性科学提供了一个学习不同的物理和社会系统的机会，发展和应用编程技能，并思考科学哲学的基本问题。

通过阅读本书和做练习，你将有机会探索在其他情况下可能不会遇到的主题和想法，用 Python 练习编程，并学习更多关于数据结构和算法的知识。

这本书的特点包括：

大多数关于复杂性科学的书都是为大众读者而写的。他们忽略了技术细节，这让那些能够处理它们的人感到沮丧。这本书展示了代码、数学以及理解模型如何工作所需的解释。

进一步阅读整本书，我包括进一步阅读的指南，包括原始论文(其中大部分可以通过电子方式获得)和来自维基百科和其他来源的相关文章。

关于每一章，我提供了一个 Jupyter 笔记本，其中包括章节的代码，附加的例子和动画，让你看到模型在运行。

练习和解决方案在每一章的结尾，我建议你可能需要练习的练习和解决方案。

对于本书中的大多数链接，我使用 URL 重定向。这种机制有隐藏链接目的地的缺点，但它使 url 更短，更不显眼。而且，更重要的是，它允许我不用更新书就可以更新链接。如果您发现一个断开的链接，请让我知道，我将改变重定向。

### 0.1 这本书是写给谁的？

本书的示例和支持代码是用 Python 编写的。您应该了解 Python 核心，熟悉其面向对象的特性，特别是使用和设计类。

If you are not already familiar with Python, you might want to start with *Think Python*, which is appropriate for people who have never programmed before. If you have programming experience in another language, there are many good Python books to choose from, as well as online resources.

I use NumPy, SciPy, and NetworkX throughout the book. If you are familiar with these libraries already, that's great, but I will also explain them when they appear.

I assume that the reader knows some mathematics: I use logarithms in several places, and vectors in one example. But that's about it.

## 0.2 Changes from the first edition

For the second edition, I added two chapters, one on evolution, the other on the evolution of cooperation.

In the first edition, each chapter presented background on a topic and suggested experiments the reader could run. For the second edition, I have done those experiments. Each chapter presents the implementation and results as a worked example, then suggests additional experiments for the reader.

For the second edition, I replaced some of my own code with standard libraries like NumPy and NetworkX. The result is more concise and more efficient, and it gives readers a chance to learn these libraries.

Also, the Jupyter notebooks are new. For every chapter there are two notebooks: one contains the code from the chapter, explanatory text, and exercises; the other contains solutions to the exercises.

Finally, all supporting software has been updated to Python 3 (but most of it runs unmodified in Python 2).

## 0.3 Using the code

All code used in this book is available from a Git repository on GitHub: <http://thinkcomplex.com/repo>. If you are not familiar with Git, it is a version

## 0.2 修改第十三版

---

如果您还不熟悉 Python，那么您可能希望从 **Think Python** 开始，它适合那些从未编过程序的人。如果你有其他语言的编程经验，你可以从许多好的 Python 书籍和在线资源中选择。

我在整本书中使用 NumPy、SciPy 和 NetworkX。如果你已经熟悉这些库，那很好，但是当它们出现的时候我也会解释它们。

我假设读者知道一些数学知识：我在几个地方使用对数，在一个例子中使用向量。但仅此而已。

### 0.2 与第一版的变化

在第二版中，我增加了两个章节，一个关于进化，另一个关于合作的进化。

在第一版中，每一章都介绍了一个主题的背景，并建议读者可以做一些实验。对于第二版，我已经做了那些实验。每一章都将实现和结果作为一个工作示例，然后为读者提供额外的实验。

在第二版中，我用 NumPy 和 NetworkX 这样的标准库替换了自己的一些代码。其结果是更简洁，更电子化，并给读者一个机会了解这些图书馆。

此外，“木星”(Jupyter)笔记本也是全新的。每一章都有两个笔记本：一个包含章节的代码、解释性文本和练习；另一个包含练习的解决方案。

最后，所有支持软件都更新为 Python 3(但是大部分软件在 Python 2 中运行 unmodi)。

### 0.3 使用代码

本书中使用的所有代码都可以在 GitHub 上的 Git 仓库中找到：<http://thinkcomplex.com/repo>。如果你不熟悉 Git，它是一个版本

control system that allows you to keep track of the files that make up a project. A collection of files under Git’s control is called a “repository”. GitHub is a hosting service that provides storage for Git repositories and a convenient web interface.

The GitHub homepage for my repository provides several ways to work with the code:

- You can create a copy of my repository by pressing the **Fork** button in the upper right. If you don’t already have a GitHub account, you’ll need to create one. After forking, you’ll have your own repository on GitHub that you can use to keep track of code you write while working on this book. Then you can clone the repo, which means that you copy the files to your computer.
- Or you can clone my repository without forking; that is, you can make a copy of my repo on your computer. You don’t need a GitHub account to do this, but you won’t be able to write your changes back to GitHub.
- If you don’t want to use Git at all, you can download the files in a Zip file using the green button that says “Clone or download”.

I developed this book using Anaconda from Continuum Analytics, which is a free Python distribution that includes all the packages you’ll need to run the code (and lots more). I found Anaconda easy to install. By default it does a user-level installation, not system-level, so you don’t need administrative privileges. And it supports both Python 2 and Python 3. You can download Anaconda from <https://continuum.io/downloads>.

The repository includes both Python scripts and Jupyter notebooks. If you have not used Jupyter before, you can read about it at <https://jupyter.org>.

There are three ways you can work with the Jupyter notebooks:

**Run Jupyter on your computer** If you installed Anaconda, you can install Jupyter by running the following command in a terminal or Command Window:

```
$ conda install jupyter
```

控制系统，让你跟踪组成一个项目的 `les`。在 `Git` 的控制下的 `les` 集合称为“仓库”。是一个托管服务，为 `Git` 存储库和方便的 `web` 界面提供存储空间。

我的仓库的 `GitHub` 主页提供了几种使用代码的方法:

您可以通过按下右上角的 `Fork` 按钮来创建我的存储库的副本。如果你还没有 `GitHub` 帐号，你需要创建一个。分叉之后，你将在 `GitHub` 上拥有自己的存储库，可以用它来跟踪你在写这本书时编写的代码。然后你可以克隆回购，这意味着你复制的 `les`

到你的电脑上。

或者，您可以克隆我的存储库而无需分叉；也就是说，您可以在您的计算机上制作我的回购的副本。你不需要一个 `GitHub` 帐号就可以做到这一点，但是你不能把你的修改写回 `GitHub`。

如果您根本不想使用 `Git`，您可以使用绿色的克隆或下载按钮来下载 `Zip` 中的 `les`。

我使用 `Continuum Analytics` 的 `Anaconda` 开发了这本书，这是一个免费的 `Python` 发行版，其中包括运行代码所需的所有包(以及更多)。我发现蟒蛇很容易安装。默认情况下，它执行用户级安装，而不是系统级安装，因此您不需要管理特权。并且它支持 `Python 2` 和 `Python 3`。你可以从 <https://continuum.io/downloads> 下载蟒蛇。

该存储库包括 `Python` 脚本和 `Jupyter` 笔记本。如果你以前没有使用过木星，你可以在 <https://Jupyter.org> 网站上阅读相关内容。

你可以通过三种方式使用 `Jupyter` 笔记本:

在你的电脑上运行 `Jupyter` 如果你安装了 `Anaconda`，你可以通过在终端或命令窗口中运行以下命令来安装 `Jupyter`:

```
$conda install jupyter
```

Before you launch Jupyter, you should `cd` into the directory that contains the code:

```
$ cd ThinkComplexity2/code
```

And then start the Jupyter server:

```
$ jupyter notebook
```

When you start the server, it should launch your default web browser or create a new tab in an open browser window. Then you can open and run the notebooks.

**Run Jupyter on Binder** Binder is a service that runs Jupyter in a virtual machine. If you follow this link, <http://thinkcomplex.com/binder>, you should get a Jupyter home page with the notebooks for this book and the supporting data and scripts.

You can run the scripts and modify them to run your own code, but the virtual machine you run them in is temporary. If you leave it idle, the virtual machine disappears along with any changes you made.

**View notebooks on GitHub** GitHub provides a view of the notebooks you can use to read the notebooks and see the results I generated, but you won't be able to modify or run the code.

Good luck, and have fun!

Allen B. Downey  
Professor of Computer Science  
Olin College of Engineering  
Needham, MA

## Contributor List

If you have a suggestion or correction, please send email to [downey@allendowney.com](mailto:downey@allendowney.com). If I make a change based on your feedback, I will add you to the contributor list (unless you ask to be omitted).

Let me know what version of the book you are working with, and what format. If you include at least part of the sentence the error appears in, that makes it easy for me to search. Page and section numbers are fine, too, but not quite as easy to work with. Thanks!

在你启动 Jupyter 之前，你应该进入包含以下代码的目录：

```
$cd ThinkComplexity2/code
```

然后启动 Jupyter 服务器：

```
$jupyter notebook
```

当你启动服务器时，它应该启动你的默认浏览器或者在打开的浏览器窗口中创建一个新的标签页。然后你可以打开并运行这些笔记本。

在 Binder 上运行 Jupyter 是一个在虚拟机上运行 Jupyter 的服务。如果你点击这个链接，<http://thinkcomplex.com/binder>，你会得到一个 Jupyter 主页，里面有这本书的笔记本以及相关的数据和脚本。

您可以运行这些脚本并修改它们以运行您自己的代码，但是在其中运行它们的虚拟机是临时的。如果将其闲置，则虚拟机将随着所做的任何更改一起消失。

上的视图笔记本提供了一个视图，你可以用它来阅读笔记本并查看我生成的结果，但是你不能修改或运行代码。

祝你好运，玩得开心！

艾伦 ·b·唐尼  
计算机科学教授  
奥林工程学院  
李约瑟

投稿人名单

如果你有什么建议或者更正，请发送电子邮件到 [downey@alltarpney.com](mailto:downey@alltarpney.com)。如果我根据您的反馈进行了更改，我将把你添加到贡献者列表中(除非您要求省略)。

让我知道什么版本的书籍你正在工作，什么格式。如果你至少包含出现错误的句子的一部分，那么我就很容易搜索。页码和部分号码也是一个，但不太容易使用。谢谢！

- John Harley, Jeff Stanton, Colden Rouleau and Keerthik Omanakuttan are Computational Modeling students who pointed out typos.
- Jose Oscar Mur-Miranda found several typos.
- Phillip Loh, Corey Dolphin, Noam Rubin and Julian Ceipek found typos and made helpful suggestions.
- Sebastian Schöner sent two pages of corrections!
- Philipp Marek sent a number of corrections.
- Jason Woodard co-taught Complexity Science with me at Olin College, introduced me to NK models, and made many helpful suggestions and corrections.
- Davi Post sent several corrections and suggestions.
- Graham Taylor sent a pull request on GitHub that fixed many typos.

I would especially like to thank the technical reviewers, Vincent Knight and Eric Ma, who made many helpful suggestions, and the copy editor, Charles Roumeliotis, who caught many errors and inconsistencies.

Other people who reported errors include Richard Hollands, Muhammad Najmi bin Ahmad Zabidi, Alex Hantman, and Jonathan Harford.

约翰·哈利、杰·斯坦顿、科尔登·鲁洛和基尔蒂克·奥马纳库坦是指出拼写错误的计算模型专业的学生。

何塞·奥斯卡·穆尔-米兰达发现了几个拼写错误。

菲利普·洛、科里·多芬、诺姆·鲁宾和朱利安·塞佩克发现了错误，并提出了有益的建议。

塞巴斯蒂安·舒纳送来了两页更正！

菲利普·马雷克做了一些更正。

Jason Woodard 和我一起<sup>1</sup>在奥林学院教授复杂性科学，向我介绍了 NK 模型，并提出了许多有益的建议和修正。

大卫·邮报发送了一些更正和建议。

格雷厄姆·泰勒在 GitHub 上发送了一个拉取请求，这个请求删除了许多打印错误。

我要特别感谢技术评论家 Vincent Knight 和 Eric Ma，他们提出了许多有益的建议，还有文字编辑 Charles Roumeliotis，他们发现了许多错误和不一致的地方。

其他报告错误的人包括 Richard Hollands，Muhammad Najmi bin Ahmad Zabidi，Alex Hantman 和 Jonathan Harford。

# Chapter 1

## Complexity Science

Complexity science is relatively new; it became recognizable as a field, and was given a name, in the 1980s. But its newness is not because it applies the tools of science to a new subject, but because it uses different tools, allows different kinds of work, and ultimately changes what we mean by “science”.

To demonstrate the difference, I’ll start with an example of classical science: suppose someone asks you why planetary orbits are elliptical. You might invoke Newton’s law of universal gravitation and use it to write a differential equation that describes planetary motion. Then you can solve the differential equation and show that the solution is an ellipse. QED!

Most people find this kind of explanation satisfying. It includes a mathematical derivation — so it has some of the rigor of a proof — and it explains a specific observation, elliptical orbits, by appealing to a general principle, gravitation.

Let me contrast that with a different kind of explanation. Suppose you move to a city like Detroit that is racially segregated, and you want to know why it’s like that. If you do some research, you might find a paper by Thomas Schelling called “Dynamic Models of Segregation”, which proposes a simple model of racial segregation:

Here is my description of the model, from Chapter 9:

The Schelling model of the city is an array of cells where each cell represents a house. The houses are occupied by two kinds of

## 第一章

### 复杂性科学

复杂性科学是相对较新的科学，它在 20 世纪 80 年代被公认为一个领域，并被赋予了一个名称。但它的新颖并不是因为它将科学的工具应用到一个新的主题上，而是因为它使用了不同的工具，允许不同类型的工作，并最终改变了我们对科学的定义”。

为了证明这一点，我将从一个古典科学的例子开始：假设有人问你为什么行星轨道是椭圆的。你可以援引牛顿的万有引力定律，用它来写一个描述行星运动的微分方程。然后求解微分方程，证明解是椭圆。QED！

大多数人对这种解释感到满意。它包括一个数学推导 | 所以它有一些严谨的证明 | 和它解释了一个特殊的观察，椭圆轨道，通过诉诸一般原理，引力。

让我把它和一种不同的解释进行对比。假设你搬到底特律这样种族隔离的城市，你想知道为什么会这样。如果你做一些研究，你可能会看到托马斯·克罗姆比·谢林的一篇名为种族隔离动态模型的论文，它提出了一个简单的种族隔离模型：

下面是我对这个模型的描述，摘自第 9 章：

城市的谢林模型是一个细胞阵列，每个细胞代表一个房子。这些房子住着两种人

“agents”, labeled red and blue, in roughly equal numbers. About 10% of the houses are empty.

At any point in time, an agent might be happy or unhappy, depending on the other agents in the neighborhood. In one version of the model, agents are happy if they have at least two neighbors like themselves, and unhappy if they have one or zero.

The simulation proceeds by choosing an agent at random and checking to see whether it is happy. If so, nothing happens; if not, the agent chooses one of the unoccupied cells at random and moves.

If you start with a simulated city that is entirely unsegregated and run the model for a short time, clusters of similar agents appear. As time passes, the clusters grow and coalesce until there are a small number of large clusters and most agents live in homogeneous neighborhoods.

The degree of segregation in the model is surprising, and it suggests an explanation of segregation in real cities. Maybe Detroit is segregated because people prefer not to be greatly outnumbered and will move if the composition of their neighborhoods makes them unhappy.

Is this explanation satisfying in the same way as the explanation of planetary motion? Many people would say not, but why?

Most obviously, the Schelling model is highly abstract, which is to say not realistic. So you might be tempted to say that people are more complicated than planets. But that can't be right. After all, some planets have people on them, so they have to be more complicated than people.

Both systems are complicated, and both models are based on simplifications. For example, in the model of planetary motion we include forces between the planet and its sun, and ignore interactions between planets. In Schelling's model, we include individual decisions based on local information, and ignore every other aspect of human behavior.

But there are differences of degree. For planetary motion, we can defend the model by showing that the forces we ignore are smaller than the ones we include. And we can extend the model to include other interactions and

大约有 10% 的房子是空的。

在任何时候，经纪人可能是高兴或不高兴，这取决于其他经纪人在附近。在这个模型的一个版本中，如果至少有两个像自己一样的邻居，代理人会感到高兴；如果只有一个或零个邻居，代理人会感到不高兴。

模拟通过随机选择一个代理并检查它是否满意来进行。如果是这样，什么也不会发生；如果不是这样，代理随机选择一个空闲的单元格并移动。

如果你从一个完全没有隔离的模拟城市开始，然后运行模型很短时间，就会出现类似的代理集群。随着时间的推移，这些集群不断成长和合并，直到出现少数大型集群，而且大多数代理都生活在同质的社区中。

这个模型中的种族隔离程度令人惊讶，并且它提出了一个真实城市中种族隔离的解释。也许底特律之所以实行种族隔离是因为人们不希望在数量上处于劣势，如果社区的构成让他们不开心，他们就会搬走。

这个解释和行星运动的解释一样令人满意吗？很多人会说没有，但是为什么呢？

最明显的是，谢林模型是高度抽象的，也就是说不现实。所以你可能会说人类比行星更复杂。但这不可能是正确的。毕竟，有些行星上有人，所以它们必须比人类更复杂。

这两个系统都是复杂的，而且两个模型都是基于简化的。例如，在行星运动的模型中，我们包括了行星和它的太阳之间的力，而忽略了行星之间的相互作用。在谢林的模型中，我们包括了基于当地信息的个人决策，而忽略了人类行为的其他方面。

但也存在程度的差异。对于行星的运动，我们可以通过证明我们忽略的力比我们包含的力小来捍卫这个模型。我们可以扩展这个模型来包括其他的交互和

show that the effect is small. For Schelling's model it is harder to justify the simplifications.

Another difference is that Schelling's model doesn't appeal to any physical laws, and it uses only simple computation, not mathematical derivation. Models like Schelling's don't look like classical science, and many people find them less compelling, at least at first. But as I will try to demonstrate, these models do useful work, including prediction, explanation, and design. One of the goals of this book is to explain how.

## 1.1 The changing criteria of science

Complexity science is not just a different set of models; it is also a gradual shift in the criteria models are judged by, and in the kinds of models that are considered acceptable.

For example, classical models tend to be law-based, expressed in the form of equations, and solved by mathematical derivation. Models that fall under the umbrella of complexity are often rule-based, expressed as computations, and simulated rather than analyzed.

Not everyone finds these models satisfactory. For example, in *Sync*, Steven Strogatz writes about his model of spontaneous synchronization in some species of fireflies. He presents a simulation that demonstrates the phenomenon, but then writes:

I repeated the simulation dozens of times, for other random initial conditions and for other numbers of oscillators. Sync every time. [...] The challenge now was to prove it. Only an ironclad proof would demonstrate, in a way that no computer ever could, that sync was inevitable; and the best kind of proof would clarify *why* it was inevitable.

Strogatz is a mathematician, so his enthusiasm for proofs is understandable, but his proof doesn't address what is, to me, the most interesting part of the phenomenon. In order to prove that "sync was inevitable", Strogatz makes several simplifying assumptions, in particular that each firefly can see all the others.

对于谢林的模型来说，要证明简单化方程式的正确性是很困难的。

另一个不同之处在于，谢林的模型不符合任何物理定律，它只使用简单的计算，而不是数学推导。像谢林这样的模型看起来并不像古典科学，而且许多人认为它们不那么令人信服，至少在一开始是这样。但是，正如我将要展示的，这些模型做了有用的工作，包括预测、解释和设计。这本书的目的之一就是解释。

### 1.1 科学标准的变化

复杂性科学不仅仅是一套不同的模型，它也是判断模型的标准和被认为是可接受的模型种类的逐渐转变。

例如，经典模型倾向于以法律为基础，以方程的形式表示，并通过数学推导求解。属于复杂性范畴的模型通常是基于规则的，表示为计算，模拟而不是分析。

并不是每个人都对这些模型感到满意。例如，在 *Sync* 中，Steven Strogatz 描述了他的一些种类的 *reies* 中的自发同步模型。他展示了一个模拟来证明这种现象，然后写道：

我重复了几十次模拟，对于其他随机的初始条件，和其他数目的振子。每次同步。[ ... ... ]现在的挑战是证明这一点。只有无懈可击的证据才能证明同步是不可避免的，这是任何计算机都无法做到的；而最好的证据就能说明为什么同步是不可避免的。

斯托加茨是一位数学家，所以他对证明的热情是可以理解的，但是他的证明并没有解决对我来说这个现象中最有趣的部分。为了证明同步是不可避免的”，斯特罗加茨做了几个简化假设，特别是每个测试都能看到其他所有测试。

In my opinion, it is more interesting to explain how an entire valley of fireflies can synchronize *despite the fact that they cannot all see each other*. How this kind of global behavior emerges from local interactions is the subject of Chapter 9. Explanations of these phenomena often use agent-based models, which explore (in ways that would be difficult or impossible with mathematical analysis) the conditions that allow or prevent synchronization.

I am a computer scientist, so my enthusiasm for computational models is probably no surprise. I don't mean to say that Strogatz is wrong, but rather that people have different opinions about what questions to ask and what tools to use to answer them. These opinions are based on value judgments, so there is no reason to expect agreement.

Nevertheless, there is rough consensus among scientists about which models are considered good science, and which others are fringe science, pseudoscience, or not science at all.

A central thesis of this book is that the criteria this consensus is based on change over time, and that the emergence of complexity science reflects a gradual shift in these criteria.

## 1.2 The axes of scientific models

I have described classical models as based on physical laws, expressed in the form of equations, and solved by mathematical analysis; conversely, models of complex systems are often based on simple rules and implemented as computations.

We can think of this trend as a shift over time along two axes:

**Equation-based** → **simulation-based**

**Analysis** → **computation**

Complexity science is different in several other ways. I present them here so you know what's coming, but some of them might not make sense until you have seen the examples later in the book.

## 4 第一章复杂性科学

---

在我看来，更有趣的是解释一个完整的 reies 山谷是如何同步的，尽管事实上他们不能看到彼此。这种全局行为是如何从局部相互作用中产生的，这是第九章的主题。对这些现象的解释通常使用基于代理的模型，这些模型探索(用数学分析是不可能或不可能的方式)允许或阻止同步的条件。

我是一名计算机科学家，所以我对计算机模型的热情可能并不令人惊讶。我并不是说斯托加茨是错误的，而是人们对于该问什么问题以及用什么工具来回答这些问题有着截然不同的意见。这些观点是基于价值判断的，所以没有理由期待达成一致。

然而，对于哪些模型被认为是好的科学，哪些是边缘科学，伪科学，或者根本不是科学，科学家们有着粗略的共识。

这本书的中心论点是，这一共识的标准是基于随着时间的推移而变化的，复杂性科学的出现反映了这些标准的逐渐转变。

### 1.2 科学模型的轴线

我曾将经典模型描述为基于物理定律，以方程的形式表示，并通过数学分析求解；相反，复杂系统的模型通常基于简单的规则，并以计算的形式实现。

我们可以把这种趋势看作是随着时间的推移沿着两条轴线的转变：

基于方程的! 基于模拟的  
分析! 计算

复杂性科学在其他几个方面是不同的。我把它们放在这里，这样你们就知道接下来会发生什么，但是其中的一些可能不会有意义，直到你们在本书后面看到这些例子。

**Continuous** → **discrete** Classical models tend to be based on continuous mathematics, like calculus; models of complex systems are often based on discrete mathematics, including graphs and cellular automata.

**Linear** → **nonlinear** Classical models are often linear, or use linear approximations to nonlinear systems; complexity science is more friendly to nonlinear models.

**Deterministic** → **stochastic** Classical models are usually deterministic, which may reflect underlying philosophical determinism, discussed in Chapter 5; complex models often include randomness.

**Abstract** → **detailed** In classical models, planets are point masses, planes are frictionless, and cows are spherical (see <http://thinkcomplex.com/cow>). Simplifications like these are often necessary for analysis, but computational models can be more realistic.

**One, two** → **many** Classical models are often limited to small numbers of components. For example, in celestial mechanics the two-body problem can be solved analytically; the three-body problem cannot. Complexity science often works with large numbers of components and larger number of interactions.

**Homogeneous** → **heterogeneous** In classical models, the components and interactions tend to be identical; complex models more often include heterogeneity.

These are generalizations, so we should not take them too seriously. And I don't mean to deprecate classical science. A more complicated model is not necessarily better; in fact, it is usually worse.

And I don't mean to say that these changes are abrupt or complete. Rather, there is a gradual migration in the frontier of what is considered acceptable, respectable work. Some tools that used to be regarded with suspicion are now common, and some models that were widely accepted are now regarded with scrutiny.

For example, when Appel and Haken proved the four-color theorem in 1976, they used a computer to enumerate 1,936 special cases that were, in some sense, lemmas of their proof. At the time, many mathematicians did not consider

连续的！离散经典模型倾向于基于连续数学，如微积分；复杂系统的模型通常基于离散数学，包括图形和元胞自动机。

线性的！非线性经典模型往往是线性的，或使用线性逼近非线性系统；复杂性科学是更友好的非线性模型。

确定性！随机经典模型通常是确定性的，这可能反映了潜在的哲学决定论，在第 5 章讨论；复杂的模型通常包括随机性。

抽象！在经典模型中，行星是点质量，平面是无摩擦的，母牛是球形的(见 <http://thinkcomplex.com/>)。像这样的简单公式通常是分析所必需的，但是计算模型可以更加真实。

一，二！许多经典模型往往局限于少量的组件。例如，在 21 世纪天体力学，二体问题可以通过分析求解，而三体不能。复杂性科学通常与大量的组件和大量的相互作用一起工作。

同质的！在经典模型中，成分和相互作用往往是相同的，复杂模型往往包括异质性。

这些都是概括性的，我们不应该太当真。我并不是要反对古典科学。更复杂的模型并不一定更好；事实上，它通常更糟糕。

我并不是说这些变化是突然的或完全的。相反，在那些被认为是可以接受的、受人尊敬的工作的前沿，有一种逐渐的迁移。一些曾经被怀疑的工具现在很常见，一些曾经被广泛接受的模型现在被仔细检查。

例如，当阿佩尔和哈肯在 1976 年证明四色定理时，他们用计算机列举了 1936 个特殊情况，在某种意义上，这些情况是他们证明的引理。当时，许多数学家没有考虑到

the theorem truly proved. Now computer-assisted proofs are common and generally (but not universally) accepted.

Conversely, a substantial body of economic analysis is based on a model of human behavior called “Economic man”, or, with tongue in cheek, *Homo economicus*. Research based on this model was highly regarded for several decades, especially if it involved mathematical virtuosity. More recently, this model is treated with skepticism, and models that include imperfect information and bounded rationality are hot topics.

### 1.3 Different models for different purposes

Complex models are often appropriate for different purposes and interpretations:

**Predictive** → **explanatory** Schelling’s model of segregation might shed light on a complex social phenomenon, but it is not useful for prediction. On the other hand, a simple model of celestial mechanics can predict solar eclipses, down to the second, years in the future.

**Realism** → **instrumentalism** Classical models lend themselves to a realist interpretation; for example, most people accept that electrons are real things that exist. Instrumentalism is the view that models can be useful even if the entities they postulate don’t exist. George Box wrote what might be the motto of instrumentalism: “All models are wrong, but some are useful.”

**Reductionism** → **holism** Reductionism is the view that the behavior of a system can be explained by understanding its components. For example, the periodic table of the elements is a triumph of reductionism, because it explains the chemical behavior of elements with a model of electrons in atoms. Holism is the view that some phenomena that appear at the system level do not exist at the level of components, and cannot be explained in component-level terms.

We get back to explanatory models in Chapter 4, instrumentalism in Chapter 6, and holism in Chapter 8.

现在计算机辅助的证明已经很普遍了，并且被普遍接受(但不是普遍接受)。

相反，大量的经济分析都是基于一种被称为“经济人”的人类行为模型，或者半开玩笑地称为“经济人”。基于这个模型的研究几十年来备受推崇，特别是如果它涉及到数学精湛技巧的话。最近，这个模型受到了质疑，包括不完美信息和有限理性的模型是热门话题。

### 1.3 不同目的的不同模型

复杂模型通常适用于不同的目的和解释：

**预测！** 解释性的谢林的种族隔离模型可能阐明了一个复杂的社会现象，但它对预测没有用处。另一方面，一个简单的天体力学模型可以预测到未来的第二年的日食。

**现实主义！** 工具主义经典模型给出了一个现实主义的解释；例如，大多数人接受电子是真实存在的东西。工具主义认为，即使模型所假设的实体并不存在，它也是有用的。乔治·博克斯(George Box)写下了可能是工具主义的座右铭：所有模型都是错误的，但有些是有用的。”

**简化论！** 整体论还原论是认为一个系统的行为可以通过理解其组成部分来解释的观点。例如，元素的元素周期表是还原论的胜利，因为它用原子中的电子模型来解释元素的化学行为。整体论是这样一种观点，即在系统级别出现的某些现象并不存在于组件级别，并且不能用组件级别的术语来解释。

我们回到第四章的解释模型，第六章的工具主义，第八章的整体主义。

## 1.4 Complexity engineering

I have been talking about complex systems in the context of science, but complexity is also a cause, and effect, of changes in engineering and the design of social systems:

**Centralized** → **decentralized** Centralized systems are conceptually simple and easier to analyze, but decentralized systems can be more robust. For example, in the World Wide Web clients send requests to centralized servers; if the servers are down, the service is unavailable. In peer-to-peer networks, every node is both a client and a server. To take down the service, you have to take down *every* node.

**One-to-many** → **many-to-many** In many communication systems, broadcast services are being augmented, and sometimes replaced, by services that allow users to communicate with each other and create, share, and modify content.

**Top-down** → **bottom-up** In social, political and economic systems, many activities that would normally be centrally organized now operate as grassroots movements. Even armies, which are the canonical example of hierarchical structure, are moving toward devolved command and control.

**Analysis** → **computation** In classical engineering, the space of feasible designs is limited by our capability for analysis. For example, designing the Eiffel Tower was possible because Gustave Eiffel developed novel analytic techniques, in particular for dealing with wind load. Now tools for computer-aided design and analysis make it possible to build almost anything that can be imagined. Frank Gehry's Guggenheim Museum Bilbao is my favorite example.

**Isolation** → **interaction** In classical engineering, the complexity of large systems is managed by isolating components and minimizing interactions. This is still an important engineering principle; nevertheless, the availability of computation makes it increasingly feasible to design systems with complex interactions between components.

**Design** → **search** Engineering is sometimes described as a search for solutions in a landscape of possible designs. Increasingly, the search process

### 1.4 复杂性工程

我一直在科学的背景下谈论复杂的系统，但是复杂性也是工程和社会系统设计变化的原因，等等：

**集中！**分散的集中式系统概念简单，易于分析，但分散式系统可以更健壮。例如，在万维网客户机中，向中央服务器发送请求；如果服务器宕机，则服务不可用。在对等网络中，每个节点既是客户端又是服务器。要关闭服务，必须关闭每个节点。

**一对多！**在许多通信系统中，广播服务正在得到扩展，有时被允许用户相互通信、创建、共享和修改内容的服务所取代。

**自上而下！**在社会、政治和经济制度中，许多通常由中央组织的活动现在作为基层运动运作。即使是军队，作为等级结构的典型例子，也在朝着下放指挥和控制的方向发展。

**分析！**在经典工程中，可行设计的空间受到我们分析能力的限制。例如，设计 **Ei el Tower** 是不可能的，因为 **Gustave Ei el** 开发了新颖的分析技术，特别是处理风荷载。现在，用于计算机辅助设计和分析的工具可以构建几乎任何可以想象的东西。的毕尔巴鄂古根海姆美术馆是我最喜欢的例子。

**与世隔绝！**交互在经典工程中，大系统的复杂性是通过隔离组件和最小化交互来管理的。这仍然是一个重要的工程原则；尽管如此，计算的可用性使得设计组件之间复杂交互的系统变得越来越可行。

**设计！**搜索工程有时被描述为在一系列可能的设计中寻找解决方案。越来越多的搜索过程

can be automated. For example, genetic algorithms explore large design spaces and discover solutions human engineers would not imagine (or like). The ultimate genetic algorithm, evolution, notoriously generates designs that violate the rules of human engineering.

## 1.5 Complexity thinking

We are getting farther afield now, but the shifts I am postulating in the criteria of scientific modeling are related to 20th century developments in logic and epistemology.

**Aristotelian logic** → **many-valued logic** In traditional logic, any proposition is either true or false. This system lends itself to math-like proofs, but fails (in dramatic ways) for many real-world applications. Alternatives include many-valued logic, fuzzy logic, and other systems designed to handle indeterminacy, vagueness, and uncertainty. Bart Kosko discusses some of these systems in *Fuzzy Thinking*.

**Frequentist probability** → **Bayesianism** Bayesian probability has been around for centuries, but was not widely used until recently, facilitated by the availability of cheap computation and the reluctant acceptance of subjectivity in probabilistic claims. Sharon Bertsch McGrayne presents this history in *The Theory That Would Not Die*.

**Objective** → **subjective** The Enlightenment, and philosophic modernism, are based on belief in objective truth, that is, truths that are independent of the people that hold them. 20th century developments including quantum mechanics, Gödel's Incompleteness Theorem, and Kuhn's study of the history of science called attention to seemingly unavoidable subjectivity in even "hard sciences" and mathematics. Rebecca Goldstein presents the historical context of Gödel's proof in *Incompleteness*.

**Physical law** → **theory** → **model** Some people distinguish between laws, theories, and models. Calling something a "law" implies that it is objectively true and immutable; "theory" suggests that it is subject to revision; and "model" concedes that it is a subjective choice based on simplifications and approximations.

可以自动化。例如，遗传算法探索大型设计空间，发现人类工程师无法想象(或喜欢)的解决方案。最终的遗传算法---- 进化---- 产生的设计违反了人类工程学的规则。

### 1.5 复杂性思维

我们现在正在走得更远，但是我在科学建模标准中假定的转变与 20 世纪逻辑学和认识论的发展有关。

亚里士多德逻辑学！多值逻辑在传统逻辑中，任何命题要么是对的，要么是错的。这个系统适用于类似于数学的证明，但是对于许多现实世界的应用来说(以引人注目的方式)是失败的。替代方案包括多值逻辑、模糊逻辑和其他用于处理不确定性、模糊性和不确定性的系统。Bart Kosko 在《模糊思维》中讨论了其中的一些系统。

频率概率！贝叶斯贝叶斯概率已经存在了几个世纪，但是直到最近才被广泛使用，这是由于廉价计算的可用性和不情愿地接受概率主张的主观性。莎朗·伯奇·麦克格雷恩在《不会消亡的理论》一书中介绍了这段历史。

目标！主观的启蒙运动和哲学上的现代主义，是基于对客观真理的信仰，也就是说，真理是独立于拥有它们的人的。20 世纪的发展包括

量子力学，哥德尔的不完全性定理，以及 Kuhn 对科学史的研究都呼吁人们关注看似不可避免的主观性，甚至包括自然科学和数学。丽贝卡戈尔德斯坦介绍了历史背景的哥德尔的证明在不完整性。

物理定律！理论！有些人区分法律、理论和模型。称某物为定律意味着它客观上是真实的和不可改变的；理论意味着它可以被修正；模型承认它是一个基于简化和近似的主观选择。

I think they are all the same thing. Some concepts that are called laws are really definitions; others are, in effect, the assertion that a certain model predicts or explains the behavior of a system particularly well. We come back to the nature of physical laws in Section 4.8, Section 5.9 and Section 8.8.

**Determinism** → **indeterminism** Determinism is the view that all events are caused, inevitably, by prior events. Forms of indeterminism include randomness, probabilistic causation, and fundamental uncertainty. We come back to this topic in Section 5.5 and Section 10.6

These trends are not universal or complete, but the center of opinion is shifting along these axes. As evidence, consider the reaction to Thomas Kuhn's *The Structure of Scientific Revolutions*, which was reviled when it was published and is now considered almost uncontroversial.

These trends are both cause and effect of complexity science. For example, highly abstracted models are more acceptable now because of the diminished expectation that there should be a unique, correct model for every system. Conversely, developments in complex systems challenge determinism and the related concept of physical law.

This chapter is an overview of the themes coming up in the book, but not all of it will make sense before you see the examples. When you get to the end of the book, you might find it helpful to read this chapter again.

我认为它们都是一回事。有些被称为定律的概念实际上是定律；其他的则是断言某个模型特别好地预测或解释了一个系统的行为。我们在第 4.8 节，第 5.9 节和第 8.8 节回到了物理定律的本质。

决定论！非决定论认为所有事件都不可避免地由先前的事件引起。非决定论的形式包括随机性、概率因果关系和基本的不确定性。我们在第 5.5 节和第 10.6 节回到这个主题

这些趋势并不普遍或完整，但观点的中心正在沿着这些轴线移动。作为证据，考虑一下对托马斯·库恩的《科学革命的结构》的反应，这本书一出版就遭到了谩骂，现在被认为几乎没有争议。

这些趋势既是复杂性科学发展的原因，也是复杂性科学发展的方向。例如，高度抽象的模型现在更容易被接受，因为对每个系统应该有一个唯一的、正确的模型的期望减少了。相反，复杂系统的发展对决定论和物理定律的相关概念提出了挑战。

这一章是对书中出现的主题的概述，但是在你看到这些例子之前，并不是所有的内容都有意义。当你读到这本书的结尾时，你可能会发现再读一遍这一章会有所帮助。





# Chapter 2

## Graphs

The next three chapters are about systems made up of components and connections between components. For example, in a social network, the components are people and connections represent friendships, business relationships, etc. In an ecological food web, the components are species and the connections represent predator-prey relationships.

In this chapter, I introduce NetworkX, a Python package for building models of these systems. We start with the Erdős-Rényi model, which has interesting mathematical properties. In the next chapter we move on to models that are more useful for explaining real-world systems.

The code for this chapter is in `chap02.ipynb` in the repository for this book. More information about working with the code is in Section 0.3.

### 2.1 What is a graph?

To most people a “graph” is a visual representation of data, like a bar chart or a plot of stock prices over time. That’s not what this chapter is about.

In this chapter, a **graph** is a representation of a system that contains discrete, interconnected elements. The elements are represented by **nodes** — also called **vertices** — and the interconnections are represented by **edges**.

## 第二章

### 图表

接下来的三章是关于由组件组成的系统和组件之间的连接。例如，在一个社交网络中，组成部分是人，而关系代表友谊、商业关系等。在一个生态食物网中，组成部分是物种，其间的联系代表捕食者与被捕食者的关系。

在本章中，我将介绍 `NetworkX`，一个用于构建这些系统模型的 Python 包。我们从 Erdős-Rényi 模型开始，它具有有趣的数学性质。在下一章中，我们将继续讨论对解释现实世界系统更有用的模型。

本章的代码位于本书知识库中的 `chap02.ipynb` 中。

关于使用代码的更多信息请参见 0.3 部分。

### 2.1 什么是图表？

对大多数人来说，“图表”是数据的直观表示，就像条形图或者股票价格随时间变化的图表。这不是这一章的内容。

在本章中，图是一个包含离散的、互联的元素的系统的表示。元素由节点  $v$  也称为顶点  $v$  表示，互连用边表示。

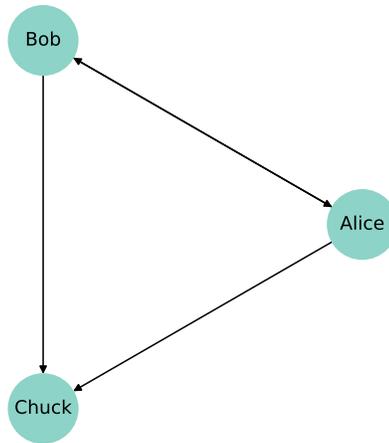


Figure 2.1: A directed graph that represents a social network.

For example, you could represent a road map with a node for each city and an edge for each road between cities. Or you could represent a social network using a node for each person, with an edge between two people if they are friends.

In some graphs, edges have attributes like length, cost, or weight. For example, in a road map, the length of an edge might represent distance between cities or travel time. In a social network there might be different kinds of edges to represent different kinds of relationships: friends, business associates, etc.

Edges may be **directed** or **undirected**, depending on whether the relationships they represent are asymmetric or symmetric. In a road map, you might represent a one-way street with a directed edge and a two-way street with an undirected edge. In some social networks, like Facebook, friendship is symmetric: if  $A$  is friends with  $B$  then  $B$  is friends with  $A$ . But on Twitter, for example, the “follows” relationship is not symmetric; if  $A$  follows  $B$ , that doesn’t imply that  $B$  follows  $A$ . So you might use undirected edges to represent a Facebook network and directed edges for Twitter.

Graphs have interesting mathematical properties, and there is a branch of mathematics called **graph theory** that studies them.

Graphs are also useful, because there are many real world problems that can be solved using **graph algorithms**. For example, Dijkstra’s shortest path algorithm is an efficient way to find the shortest path from a node to all other

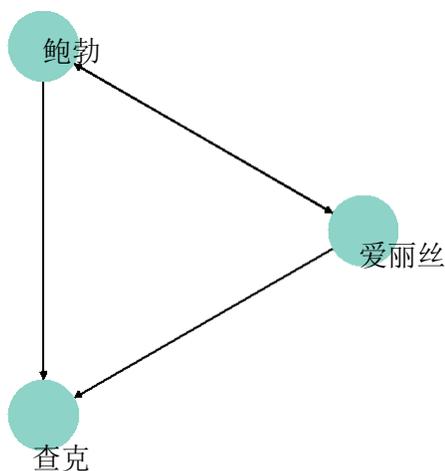


图 2.1: 表示社交网络的有向图。

例如，您可以表示一个路线图，其中包含每个城市的节点和城市之间每条道路的边界。或者你可以用每个人的一个节点来表示一个社交网络，如果他们是朋友的话，两个人之间有一个边。

在一些图中，边具有长度、成本或权重等属性。例如，在路线图中，边缘的长度可能代表城市之间的距离或旅行时间。在一个社交网络中，可能有不同类型的边来代表不同种类的关系：朋友、商业伙伴等等。

边可以是有向的或无向的，这取决于它们所代表的关系是不对称的还是对称的。在路线图中，你可以用有向边界来表示单行道，用无向边界来表示双行道。在一些社交网络中，比如 Facebook，友谊是对称的：如果 a 是 b 的朋友，那么 b 是 a 的朋友。但是在 Twitter 上，例如，跟随关系不对称；如果 a 跟随 b，这并不意味着 b 跟随 a。所以你可以使用无向边来代表一个 Facebook 网络和 Twitter 的有向边。

图具有有趣的数学性质，有一个数学分支叫做图论来研究它们。

图也很有用，因为有许多现实世界的问题可以用图算法来解决。例如，Dijkstra 的最短路径算法是一种求解从一个节点到所有其他节点的最短路径的方法

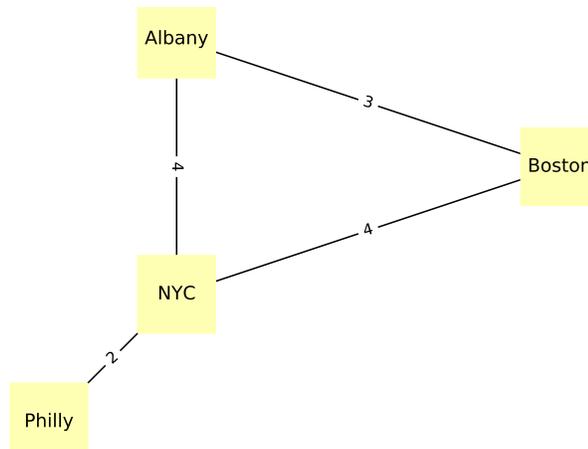


Figure 2.2: An undirected graph that represents driving time between cities.

nodes in a graph. A **path** is a sequence of nodes with an edge between each consecutive pair.

Graphs are usually drawn with squares or circles for nodes and lines for edges. For example, the directed graph in Figure 2.1 might represent three people who follow each other on Twitter. The arrow indicates the direction of the relationship. In this example, Alice and Bob follow each other, both follow Chuck, and Chuck follows no one.

The undirected graph in Figure 2.2 shows four cities in the northeast United States; the labels on the edges indicate driving time in hours. In this example the placement of the nodes corresponds roughly to the geography of the cities, but in general the layout of a graph is arbitrary.

## 2.2 NetworkX

To represent graphs, we'll use a package called NetworkX, which is the most commonly used network library in Python. You can read more about it at <http://thinkcomplex.com/netx>, but I'll explain it as we go along.

We can create a directed graph by importing NetworkX (usually imported as `nx`) and instantiating `nx.DiGraph`:

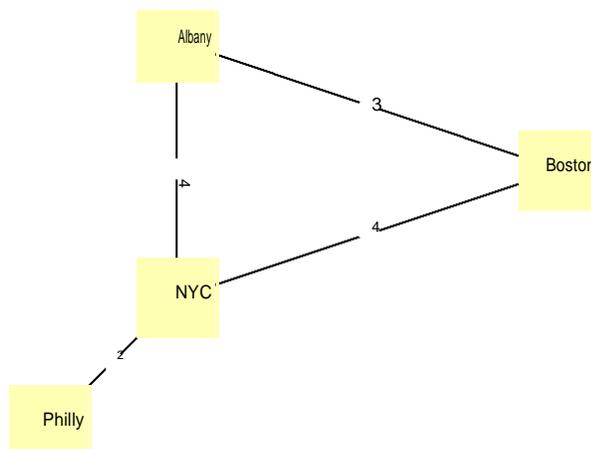


图 2.2: 一个代表城市间驾驶时间的无向图。

路径是一系列的节点，每一个连续的对之间都有一条边。

图通常是绘制正方形或圆形的节点和线的边。例如，图 2.1 中的有向图可能代表三个在 Twitter 上互相关注的人。箭头表示关系的方向。在这个例子中，Alice 和 Bob 相互跟随，都跟随 Chuck，而 Chuck 不跟随任何人。

图 2.2 中的无向图显示了美国东北部的四个城市；边缘上的标签表示以小时为单位的驾驶时间。在这个例子中，节点的位置大致对应于城市的地理位置，但是一般来说，图的布局是任意的。

## 2.2 NetworkX

为了表示图形，我们将使用一个名为 NetworkX 的包，它是 Python 中最常用的网络库。你可以在《<http://thinkcomplex.com/netx>》读到更多关于它的内容，但我会在我们继续讨论的过程中解释它。

我们可以通过导入 NetworkX (通常导入为 `nx`) 和实例化 `nx.DiGraph` 来创建一个有向图:

```
import networkx as nx
G = nx.DiGraph()
```

At this point, `G` is a `DiGraph` object that contains no nodes and no edges. We can add nodes using the `add_node` method:

```
G.add_node('Alice')
G.add_node('Bob')
G.add_node('Chuck')
```

Now we can use the `nodes` method to get a list of nodes:

```
>>> list(G.nodes())
NodeView(('Alice', 'Bob', 'Chuck'))
```

The `nodes` method returns a `NodeView`, which can be used in a for loop or, as in this example, used to make a list.

Adding edges works pretty much the same way:

```
G.add_edge('Alice', 'Bob')
G.add_edge('Alice', 'Chuck')
G.add_edge('Bob', 'Alice')
G.add_edge('Bob', 'Chuck')
```

And we can use `edges` to get the list of edges:

```
>>> list(G.edges())
[('Alice', 'Bob'), ('Alice', 'Chuck'),
 ('Bob', 'Alice'), ('Bob', 'Chuck')]
```

NetworkX provides several functions for drawing graphs; `draw_circular` arranges the nodes in a circle and connects them with edges:

```
nx.draw_circular(G,
                 node_color=COLORS[0],
                 node_size=2000,
                 with_labels=True)
```

That's the code I use to generate Figure 2.1. The option `with_labels` causes the nodes to be labeled; in the next example we'll see how to label the edges.

```
将 networkx 导入为 nx
G = nx.DiGraph ()
```

此时，`g` 是一个 `DiGraph` 对象，它不包含节点和边。我们可以使用 `add_node` 方法添加节点：

```
G.add_node (Alice)
G.add_node (Bob)
G.add_node ('Chuck')
```

现在我们可以使用 `nodes` 方法得到一个节点列表：

```
>>> list (G.nodes ())
NodeView (" Alice", " Bob", " Chuck")
```

`Nodes` 方法返回一个 `NodeView`，可以在 `for` 循环中使用，或者，如本例中所示，用于创建列表。

添加边缘的工作原理几乎相同：

```
爱丽丝, 鲍勃)
G.add_edge (Alice, 'Chuck')
G.add_edge (鲍勃, 爱丽丝)
G.add_edge ('Bob', 'Chuck')
```

我们可以使用边来得到边列表：

```
>>> 列表(G.edges ())
爱丽丝, 鲍勃, 爱丽丝, 查克,
('Bob', 'Alice'), ('Bob', 'Chuck')
```

提供了几个绘制图形的函数；`draw_round` 将节点排成一个圆，并用边连接它们：

```
绘制循环(g,
          = COLORS [0],
          2000,
          使用 labels = True)
```

To generate Figure 2.2, I start with a dictionary that maps from each city name to its approximate longitude and latitude:

```
positions = dict(Albany=(-74, 43),
                 Boston=(-71, 42),
                 NYC=(-74, 41),
                 Philly=(-75, 40))
```

Since this is an undirected graph, I instantiate `nx.Graph`:

```
G = nx.Graph()
```

Then I can use `add_nodes_from` to iterate the keys of `positions` and add them as nodes:

```
G.add_nodes_from(positions)
```

Next I'll make a dictionary that maps from each edge to the corresponding driving time:

```
drive_times = {('Albany', 'Boston'): 3,
               ('Albany', 'NYC'): 4,
               ('Boston', 'NYC'): 4,
               ('NYC', 'Philly'): 2}
```

Now I can use `add_edges_from`, which iterates the keys of `drive_times` and adds them as edges:

```
G.add_edges_from(drive_times)
```

Instead of `draw_circular`, which arranges the nodes in a circle, I'll use `draw`, which takes the position dictionary as the second parameter:

```
nx.draw(G, positions,
        node_color=COLORS[1],
        node_shape='s',
        node_size=2500,
        with_labels=True)
```

`draw` uses `positions` to determine the locations of the nodes.

To add the edge labels, we use `draw_networkx_edge_labels`:

为了生成图 2.2，我从一个字典开始，该字典将每个城市的名称映射到它的近似经纬度：

```
职位 = dict (Albany = (- 74,43) ,  
             boston = (- 71,42) ,  
             纽约 = (- 74,41) ,  
             philly = (- 75,40)
```

由于这是一个无向图，我实例化了 `nx.Graph`：

```
G = nx. Graph ()
```

然后我可以使用 `add _ nodes _ from` 来迭代位置的键，并将它们作为节点添加：

```
从(位置)添加节点
```

接下来，我将制作一个字典，将每条边与相应的驾驶时间对应起来：

```
开车时间 = {(Albany, Boston') : 3,  
            ( 奥尔巴尼, 纽约) : 4,  
            ( 波士顿, ' NYC') : 4,  
            (' NYC' , ' Philly') : 2}
```

现在我可以使用 `add edges from`，它可以迭代 `drive` 的键并将它们作为边缘添加：

```
从(驱动器时代)
```

我将使用 `draw`，它将位置字典作为第二个参数：

```
绘制(g, 位置,  
      节点颜色[1],  
      节点形状 = ' s' ,  
      2500,  
      使用 labels = True)
```

绘制使用的位置，以确定节点的位置。

为了添加边缘标签，我们使用 `draw networkxedge` 标签：

```
nx.draw_networkx_edge_labels(G, positions,  
                             edge_labels=drive_times)
```

The `edge_labels` parameter expects a dictionary that maps from each pair of nodes to a label; in this case, the labels are driving times between cities. And that's how I generated Figure 2.2.

In both of these examples, the nodes are strings, but in general they can be any hashable type.

## 2.3 Random graphs

A random graph is just what it sounds like: a graph with nodes and edges generated at random. Of course, there are many random processes that can generate graphs, so there are many kinds of random graphs.

One of the more interesting kinds is the Erdős-Rényi model, studied by Paul Erdős and Alfréd Rényi in the 1960s.

An Erdős-Rényi graph (ER graph) is characterized by two parameters:  $n$  is the number of nodes and  $p$  is the probability that there is an edge between any two nodes. See <http://thinkcomplex.com/er>.

Erdős and Rényi studied the properties of these random graphs; one of their surprising results is the existence of abrupt changes in the properties of random graphs as random edges are added.

One of the properties that displays this kind of transition is connectivity. An undirected graph is **connected** if there is a path from every node to every other node.

In an ER graph, the probability that the graph is connected is very low when  $p$  is small and nearly 1 when  $p$  is large. Between these two regimes, there is a rapid transition at a particular value of  $p$ , denoted  $p^*$ .

Erdős and Rényi showed that this critical value is  $p^* = (\ln n)/n$ , where  $n$  is the number of nodes. A random graph,  $G(n, p)$ , is unlikely to be connected if  $p < p^*$  and very likely to be connected if  $p > p^*$ .

To test this claim, we'll develop algorithms to generate random graphs and check whether they are connected.

```

绘制网络/边缘/标签(g, 位置,
                        标签 = 驱动时间)

```

`Edge _ labels` 参数需要一个字典, 该字典将每对节点映射到一个标签; 在这种情况下, 标签是城市之间的驾驶时间。这就是我如何生成图 2.2 的。

在这两个示例中, 节点都是字符串, 但一般来说, 它们可以是任何 `hashable` 类型。

### 2.3 随机图

一个随机图就像它听起来的那样: 一个随机生成节点和边的图。当然, 有许多随机过程可以生成图, 所以有许多种随机图。

一个更有趣的类型是 Erd } os-Renyi 模型, 由 Paul Erd } os 和 Alfred Renyi 在 20 世纪 60 年代研究。

一个 Erd } os-Renyi 图(ER graph)有两个拥有属性:  $n$  是节点数,  $p$  是任意两个节点之间有边的概率。参见 <http://thinkcomplex.com/er>。

Erd } os 和 Renyi 研究了这些随机图的性质, 他们令人惊讶的结果之一是随机图的性质随着随机边的加入而发生突变。

显示这种转换的属性之一是连接性。如果存在从每个节点到另一个节点的路径, 则无向图是连通的。

在 ER 图中, 当  $p$  小时图连通的概率很低, 当  $p$  大时图连通的概率接近 1。在这两种状态之间, 存在一个特定值  $p$  的快速转换, 表示  $p$ 。

Erd } os 和 Renyi 证明了这个临界值是  $p = (\ln n) / n$ , 其中  $n$  是节点数。一个随机图  $g(n; p)$ , 当  $p < p$  时不可能连通, 当  $p > p$  时很可能连通。

为了验证这一说法, 我们将开发算法来生成随机图并检查它们是否连通。

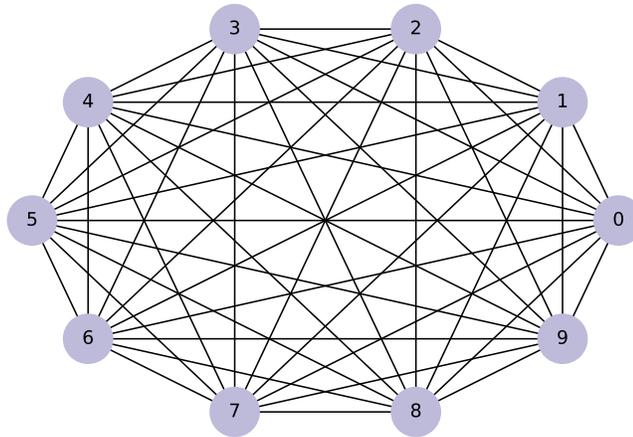


Figure 2.3: A complete graph with 10 nodes.

## 2.4 Generating graphs

I'll start by generating a **complete** graph, which is a graph where every node is connected to every other.

Here's a generator function that takes a list of nodes and enumerates all distinct pairs. If you are not familiar with generator functions, you can read about them at <http://thinkcomplex.com/gen>.

```
def all_pairs(nodes):
    for i, u in enumerate(nodes):
        for j, v in enumerate(nodes):
            if i > j:
                yield u, v
```

We can use `all_pairs` to construct a complete graph:

```
def make_complete_graph(n):
    G = nx.Graph()
    nodes = range(n)
    G.add_nodes_from(nodes)
    G.add_edges_from(all_pairs(nodes))
    return G
```

`make_complete_graph` takes the number of nodes, `n`, and returns a new `Graph` with `n` nodes and edges between all pairs of nodes.

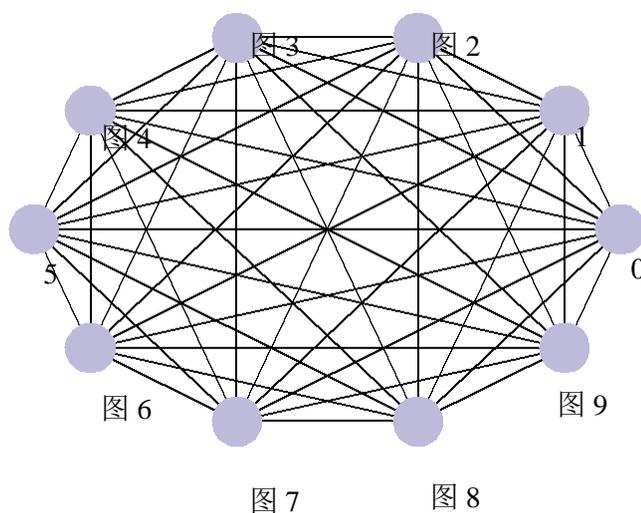


图 2.3: 一个包含 10 个节点的完整图。

## 2.4 生成图

我将从生成一个完整的图开始，这是一个每个节点都相互连接的图。

下面是一个生成器函数，它接受一个节点列表并枚举所有不同的对。如果你不熟悉生成器函数，你可以在 <http://thinkcomplex.com/gen> 上阅读它们。

```

Def all_pairs(nodes):
    对于 i, u in enumerate(nodes):
        对于枚举(节点)中的 j, v:
            如果我 > j:
                放弃 u, v

```

我们可以使用所有的对来构造一个完整的图:

```

完全图(n):
    G = nx.Graph()
    节点 = 范围(n)
    G.add_nodes_from(nodes)
    G.add_edges_from(all_pairs(nodes))
    返回 g

```

The following code makes a complete graph with 10 nodes and draws it:

```
complete = make_complete_graph(10)
nx.draw_circular(complete,
                 node_color=COLORS[2],
                 node_size=1000,
                 with_labels=True)
```

Figure 2.3 shows the result. Soon we will modify this code to generate ER graphs, but first we'll develop functions to check whether a graph is connected.

## 2.5 Connected graphs

A graph is **connected** if there is a path from every node to every other node (see <http://thinkcomplex.com/conn>).

For many applications involving graphs, it is useful to check whether a graph is connected. Fortunately, there is a simple algorithm that does it.

You can start at any node and check whether you can reach all other nodes. If you can reach a node,  $v$ , you can reach any of the **neighbors** of  $v$ , which are the nodes connected to  $v$  by an edge.

The `Graph` class provides a method called `neighbors` that returns a list of neighbors for a given node. For example, in the complete graph we generated in the previous section:

```
>>> complete.neighbors(0)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Suppose we start at node  $s$ . We can mark  $s$  as “seen” and mark its neighbors. Then we mark the neighbor’s neighbors, and their neighbors, and so on, until we can’t reach any more nodes. If all nodes are seen, the graph is connected.

Here’s what that looks like in Python:

下面的代码生成一个包含 10 个节点的完整图形，并绘制它：

```
完全 = make _ complete _ graph (10)
绘制循环(完成,
           节点颜色[2],
           1000,
           使用 labels = True)
```

图 2.3 显示了结果。很快我们将修改这段代码来生成 ER 图，但是首先我们将开发函数来检查图是否连接。

## 2.5 连通图

如果每个节点都有一条通往其他节点的路径，那么图就是连通的(参见 <http://thinkcomplex.com/conn> 图)。

对于许多涉及图的应用程序，检查图是否连通是很有用的。幸运的是，有一个简单的算法可以做到这一点。

您可以从任何节点开始，并检查是否可以到达所有其他节点。如果你可以到达一个节点， $v$ ，你可以到达  $v$  的任何邻居，这些邻居是通过边连接到  $v$  的节点。

`Graph` 类提供了一个称为“邻居”的方法，该方法返回给定节点的邻居列表。例如，在上一节生成的完整图表中：

```
>>> 完成邻居[1,2,3,4,5,6,7,8,9]
```

假设我们从节点  $s$  开始。我们可以标记  $s$  为“所见”，并标记它的邻居。然后我们标记邻居的邻居，以及他们的邻居，等等，直到我们不能到达更多的节点。如果可以看见所有节点，则该图是连通的。

下面是 Python 中的代码：

```
def reachable_nodes(G, start):
    seen = set()
    stack = [start]
    while stack:
        node = stack.pop()
        if node not in seen:
            seen.add(node)
            stack.extend(G.neighbors(node))
    return seen
```

`reachable_nodes` takes a `Graph` and a starting node, `start`, and returns the set of nodes that can be reached from `start`.

Initially the set, `seen`, is empty, and we create a list called `stack` that keeps track of nodes we have discovered but not yet processed. Initially the stack contains a single node, `start`.

Now, each time through the loop, we:

1. Remove one node from the stack.
2. If the node is already in `seen`, we go back to Step 1.
3. Otherwise, we add the node to `seen` and add its neighbors to the stack.

When the stack is empty, we can't reach any more nodes, so we break out of the loop and return `seen`.

As an example, we can find all nodes in the complete graph that are reachable from node 0:

```
>>> reachable_nodes(complete, 0)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Initially, the stack contains node 0 and `seen` is empty. The first time through the loop, node 0 is added to `seen` and all the other nodes are added to the stack (since they are all neighbors of node 0).

The next time through the loop, `pop` returns the last element in the stack, which is node 9. So node 9 gets added to `seen` and its neighbors get added to the stack.

可到达的节点(g, 开始):

```
看见 = set ()
```

```
[开始]
```

```
而 stack:
```

```
Node = stack.pop ()
```

```
如果节点没有出现:
```

```
Add (node)
```

```
Extend (G.neighbors (node))
```

```
返回看到
```

可到达的节点接受一个 **Graph** 和一个起始节点，开始，并返回一组可以从开始到达的节点。

最初的集合是空的，我们创建一个名为 **stack** 的列表来跟踪我们已经发现但尚未处理的节点。最初堆栈只包含一个节点，**start**。

现在，每次通过循环，我们:

1. 从堆栈中删除一个节点。
2. 如果节点已经在 **seen** 中，我们返回到第 1 步。
3. 否则，我们将节点添加到 **seen** 并将其邻居添加到堆栈中。

当堆栈是空的时候，我们不能到达更多的节点，所以我们打破循环并返回 **seen**。

作为一个例子，我们可以找到完整图中从节点 0 可到达的所有节点:

```
>>> 可达节点(complete,  
0){0,1,2,3,4,5,6,7,8,9}
```

最初，堆栈包含 0 节点，并且 **seen** 为空。第一次通过循环时，将节点 0 添加到 **seen**，并将所有其他节点添加到堆栈(因为它们都是节点 0 的邻居)。

下次通过循环时，**pop** 将返回堆栈中的最后一个元素，即节点 9。因此，节点 9 被添加到 **seen** 中，它的邻居被添加到堆栈中。

Notice that the same node can appear more than once in the stack; in fact, a node with  $k$  neighbors will be added to the stack  $k$  times. Later, we will look for ways to make this algorithm more efficient.

We can use `reachable_nodes` to write `is_connected`:

```
def is_connected(G):
    start = next(iter(G))
    reachable = reachable_nodes(G, start)
    return len(reachable) == len(G)
```

`is_connected` chooses a starting node by making a node iterator and choosing the first element. Then it uses `reachable` to get the set of nodes that can be reached from `start`. If the size of this set is the same as the size of the graph, that means we can reach all nodes, which means the graph is connected.

A complete graph is, not surprisingly, connected:

```
>>> is_connected(complete)
True
```

In the next section we will generate ER graphs and check whether they are connected.

## 2.6 Generating ER graphs

The ER graph  $G(n, p)$  contains  $n$  nodes, and each pair of nodes is connected by an edge with probability  $p$ . Generating an ER graph is similar to generating a complete graph.

The following generator function enumerates all possible edges and chooses which ones should be added to the graph:

```
def random_pairs(nodes, p):
    for edge in all_pairs(nodes):
        if flip(p):
            yield edge
```

`random_pairs` uses `flip`:

请注意，同一个节点可以在堆栈中出现不止一次；实际上，一个有  $k$  个邻居的节点将被添加到堆栈  $k$  倍。稍后，我们将寻找使这个算法更加电子化的方法。

我们可以使用可到达的节点来写 `is connected`:

```
是 _ connected (g) :  
    Start = next (iter (g))  
    可达 = 可达 _ 节点(g, 开始)  
    返回 len (reachable) == len (g)
```

通过创建一个节点迭代器并选择第一个元素来选择一个起始节点。然后，它使用可到达的节点集，可以从开始到达。如果这个集合的大小与图的大小相同，这意味着我们可以到达所有的节点，这意味着图是连通的。

毫不奇怪，一个完整的图表是相互关联的:

```
>>> 是 _ 连接(完整)  
没错
```

在下一节中，我们将生成 ER 图并检查它们是否连接。

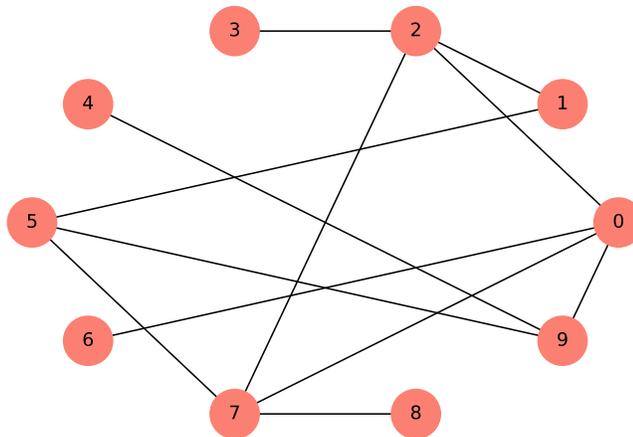
## 2.6 生成 ER 图

ER 图  $g(n; p)$  包含  $n$  个节点，每对节点通过一条概率为  $p$  的边连接，生成 ER 图类似于生成一个完全图。

下面的生成器函数列举了所有可能的边，并选择哪些边应该添加到图中:

```
返回的结果:  
    对于所有 _ 对(节点)中的边:  
        如果 flip (p) :  
            屈服边
```

随机配对使用 `flip`:

Figure 2.4: An ER graph with  $n=10$  and  $p=0.3$ .

```
def flip(p):  
    return np.random.random() < p
```

This is the first example we've seen that uses NumPy. Following convention, I import `numpy` as `np`. NumPy provides a module named `random`, which provides a method named `random`, which returns a number between 0 and 1, uniformly distributed.

So `flip` returns `True` with the given probability,  $p$ , and `False` with the complementary probability,  $1-p$ .

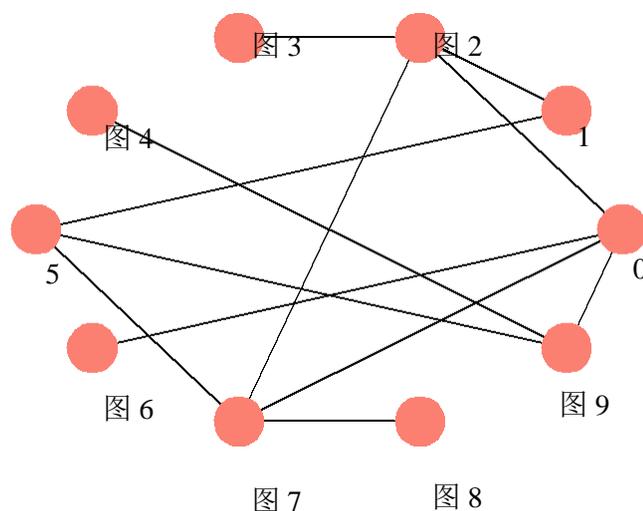
Finally, `make_random_graph` generates and returns the ER graph  $G(n, p)$ :

```
def make_random_graph(n, p):  
    G = nx.Graph()  
    nodes = range(n)  
    G.add_nodes_from(nodes)  
    G.add_edges_from(random_pairs(nodes, p))  
    return G
```

`make_random_graph` is almost identical to `make_complete_graph`; the only difference is that it uses `random_pairs` instead of `all_pairs`.

Here's an example with  $p=0.3$ :

```
random_graph = make_random_graph(10, 0.3)
```

图 2.4: 具有  $n = 10$  和  $p = 0.3$  的 ER 图。

Def flip (p) :

返回 `np.random.random() < p`

这是我们看到的第一个使用 NumPy 的例子。按照惯例，我导入 `numpy` 作为 `np`。NumPy 提供了一个名为 `random` 的模块，它提供了一个名为 `random` 的方法，该方法返回一个介于 0 和 1 之间的均匀分布的数字。

所以 `flip` 以给定的概率返回 `True`,  $p$ , 以互补的概率返回 `False`,  $1-p$ 。

最后，`make_random_graph` 生成并返回 ER 图  $g(n; p)$  :

返回一个例子:

`G = nx.Graph()`

节点 = 范围(n)

`G.add_nodes_from(nodes)`

`G.add_edges_from(random_pairs(nodes, p))`

返回 `g`

使随机图几乎等同于使完全图; 唯一的区别是它使用随机对而不是所有的对。

这里有一个  $p = 0.3$  的例子:

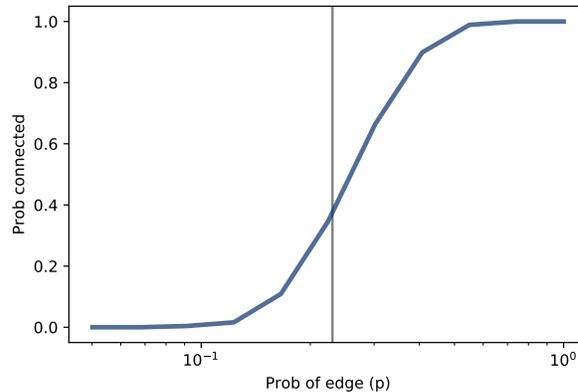


Figure 2.5: Probability of connectivity with  $n = 10$  and a range of  $p$ . The vertical line shows the predicted critical value.

Figure 2.4 shows the result. This graph turns out to be connected; in fact, most ER graphs with  $n = 10$  and  $p = 0.3$  are connected. In the next section, we'll see how many.

## 2.7 Probability of connectivity

For given values of  $n$  and  $p$ , we would like to know the probability that  $G(n, p)$  is connected. We can estimate it by generating a large number of random graphs and counting how many are connected. Here's how:

```
def prob_connected(n, p, iters=100):
    tf = [is_connected(make_random_graph(n, p))
          for i in range(iters)]
    return np.mean(tf)
```

The parameters `n` and `p` are passed along to `make_random_graph`; `iters` is the number of random graphs we generate.

This function uses a list comprehension; if you are not familiar with this feature, you can read about it at <http://thinkcomplex.com/comp>.

The result, `tf`, is a list of boolean values: `True` for each graph that's connected and `False` for each one that's not.

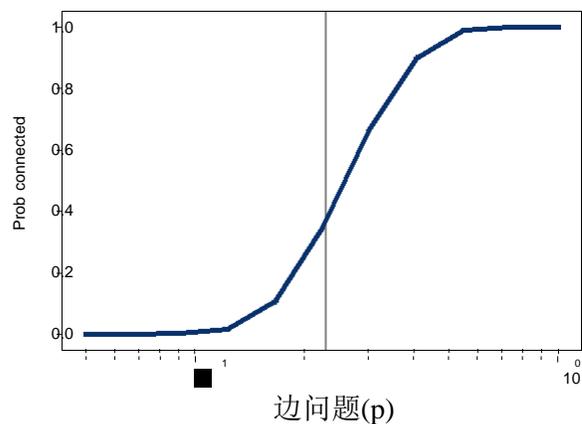


图 2.5:  $n = 10$  和  $p$  范围内连通的概率。垂直线显示了预测的临界值。

图 2.4 显示了结果。这个图被证明是连通的; 事实上, 大多数  $n = 10$  和  $p = 0.3$  的 ER 图是连通的。在下一节中, 我们将看到。

## 2.7 连接的概率

对于给定的  $n$  和  $p$  值, 我们想知道  $g(n; p)$  连通的概率。我们可以通过生成大量的随机图并计算有多少图是连通的来估计它。以下是如何做到的:

```

Def prob_connected (n, p, iters = 100):
    Tf = [ is_connected (make_random_graph (n, p))]
    因为 i 在范围内
    返回 np.mean (bool)

```

参数  $n$  和  $p$  被传递以形成随机图;  $iters$  是我们生成的随机图的个数。

这个函数使用一个列表内涵, 如果你不熟悉这个功能, 你可以在 <http://thinkcomplex.com/comp> 上阅读。

结果  $tf$  是一个布尔值列表: 对于每个连接的图为 **True**, 对于每个不连接的图为 **False**。

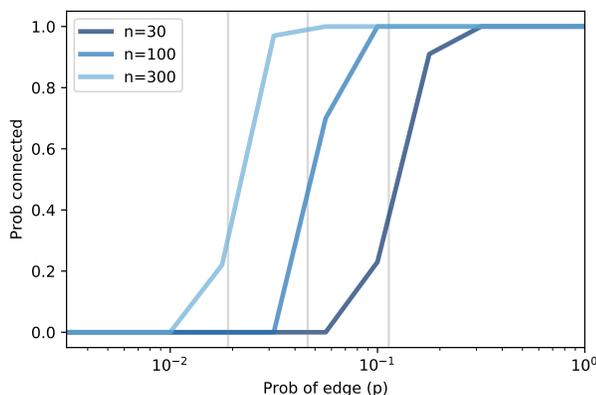


Figure 2.6: Probability of connectivity for several values of  $n$  and a range of  $p$ .

`np.mean` is a NumPy function that computes the mean of this list, treating `True` as 1 and `False` as 0. The result is the fraction of random graphs that are connected.

```
>>> prob_connected(10, 0.23, iters=10000)
0.33
```

I chose 0.23 because it is close to the critical value where the probability of connectivity goes from near 0 to near 1. According to Erdős and Rényi,  $p^* = \ln n/n = 0.23$ .

We can get a clearer view of the transition by estimating the probability of connectivity for a range of values of  $p$ :

```
n = 10
ps = np.logspace(-2.5, 0, 11)
ys = [prob_connected(n, p) for p in ps]
```

The NumPy function `logspace` returns an array of 11 values from  $10^{-2.5}$  to  $10^0 = 1$ , equally spaced on a logarithmic scale.

For each value of  $p$  in the array, we compute the probability that a graph with parameter  $p$  is connected and store the results in `ys`.

Figure 2.5 shows the results, with a vertical line at the computed critical

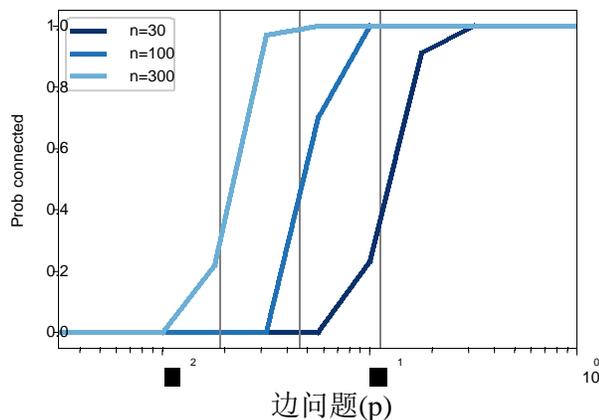


图 2.6:  $n$  值和  $p$  值范围的连通概率。

`Mean` 是一个 NumPy 函数，它计算这个列表的平均值，将 `True` 视为 1，将 `False` 视为 0。结果是连通的随机图的分值。

```
>>> Prob_connected(10,0.23, iters = 10000)
0.33
```

我选择 0.23 是因为它接近临界值，即连接的概率从接近 0 到接近 1。根据 Erdős 和 Renyi,  $p = \ln n / n = 0.23$ 。

我们可以通过估计一系列  $p$  值的连通概率来更清楚地看到这种转变:

```
10
Ps = np.logspace(-2.5,0,11)
[prob_connected(n, p) for p in ps]
```

函数 `logspace` 返回一个 11 个值的数组，从  $10^{-2.5}$  到  $10^0 = 1$ ，等间距放置在一个对数尺度上。

对于数组中的每个  $p$  值，我们计算含有参数  $p$  的图连通的概率，并将结果存储在 `y` 中。

图 2.5 显示了结果，在计算的临界点有一条垂直线

value,  $p^* = 0.23$ . As expected, the transition from 0 to 1 occurs near the critical value.

Figure 2.6 shows similar results for larger values of  $n$ . As  $n$  increases, the critical value gets smaller and the transition gets more abrupt.

These experimental results are consistent with the analytic results Erdős and Rényi presented in their papers.

## 2.8 Analysis of graph algorithms

Earlier in this chapter I presented an algorithm for checking whether a graph is connected; in the next few chapters, we will see other graph algorithms. Along the way, we will analyze the performance of those algorithms, figuring out how their run times grow as the size of the graphs increases.

If you are not already familiar with analysis of algorithms, you might want to read Appendix B of *Think Python, 2nd Edition*, at <http://thinkcomplex.com/tp2>.

The order of growth for graph algorithms is usually expressed as a function of  $n$ , the number of vertices (nodes), and  $m$ , the number of edges.

As an example, let's analyze `reachable_nodes` from Section 2.5:

```
def reachable_nodes(G, start):
    seen = set()
    stack = [start]
    while stack:
        node = stack.pop()
        if node not in seen:
            seen.add(node)
            stack.extend(G.neighbors(node))
    return seen
```

Each time through the loop, we pop a node off the stack; by default, `pop` removes and returns the last element of a list, which is a constant time operation.

值,  $p = 0.23$ 。正如预期的那样, 从 0 到 1 的转换发生在临界值附近。

图 2.6 显示了较大的  $n$  值的类似结果。随着  $n$  的增加, 临界值越来越小, 过渡越来越突兀。

这些实验结果与文献中的分析结果是一致的。

## 2.8 图形算法分析

在本章的前面, 我提出了一个检查图是否连通的算法; 在接下来的几章中, 我们将看到其他的图算法。在此过程中, 我们将分析这些算法的性能, 得出它们的运行时间是如何随着图形大小的增加而增长的。

如果你还不熟悉算法分析, 你可能想要阅读一下《Think Python, 第二版》的附录 b, <http://thinkcomplex.Com/tp2>.

图算法的增长顺序通常表示为  $n$ 、顶点(节点)数和  $m$ 、边数的函数。

作为一个例子, 让我们分析一下 2.5 节中的可达节点:

```
可到达的节点(g, 开始):
```

```
    看见 = set ()
```

```
    [开始]
```

```
    而 stack:
```

```
        Node = stack.pop ()
```

```
        如果节点没有出现:
```

```
            Add (node)
```

```
            Extend (G.neighbors (node))
```

```
    返回看到
```

每次通过循环时, 我们在堆栈中弹出一个节点; 默认情况下, 弹出删除并返回列表的最后一个元素, 这是一个常量时间操作。

Next we check whether the node is in `seen`, which is a set, so checking membership is constant time.

If the node is not already in `seen`, we add it, which is constant time, and then add the neighbors to the `stack`, which is linear in the number of neighbors.

To express the run time in terms of  $n$  and  $m$ , we can add up the total number of times each node is added to `seen` and `stack`.

Each node is only added to `seen` once, so the total number of additions is  $n$ .

But nodes might be added to `stack` many times, depending on how many neighbors they have. If a node has  $k$  neighbors, it is added to `stack`  $k$  times. Of course, if it has  $k$  neighbors, that means it is connected to  $k$  edges.

So the total number of additions to `stack` is the total number of edges,  $m$ , doubled because we consider every edge twice.

Therefore, the order of growth for this function is  $O(n + m)$ , which is a convenient way to say that the run time grows in proportion to either  $n$  or  $m$ , whichever is bigger.

If we know the relationship between  $n$  and  $m$ , we can simplify this expression. For example, in a complete graph the number of edges is  $n(n - 1)/2$ , which is in  $O(n^2)$ . So for a complete graph, `reachable_nodes` is quadratic in  $n$ .

## 2.9 Exercises

The code for this chapter is in `chap02.ipynb`, which is a Jupyter notebook in the repository for this book. For more information about working with this code, see Section 0.3.

**Exercise 2.1** Launch `chap02.ipynb` and run the code. There are a few short exercises embedded in the notebook that you might want to try.

**Exercise 2.2** In Section 2.8 we analyzed the performance of `reachable_nodes` and classified it in  $O(n + m)$ , where  $n$  is the number of nodes and  $m$  is the number of edges. Continuing the analysis, what is the order of growth for `is_connected`?

接下来我们检查节点是否在 `seen` 中，这是一个集合，因此检查成员关系是常量时间。

如果节点还没有被看到，我们添加它，这是一个常量时间，然后将邻居添加到堆栈中，这是邻居数量的线性关系。

为了用  $n$  和  $m$  来表示运行时间，我们可以将每个节点被添加到 `seen` 和 `stack` 中的总次数相加。

每个节点只被添加到 `seen` 中一次，因此添加的总数是  $n$ 。

但是节点可以多次添加到堆栈中，这取决于它们有多少个邻居。如果一个节点有  $k$  个邻居，那么它将被添加到堆栈  $k$  次。当然，如果它有  $k$  个邻居，那就意味着它连接到  $k$  个边。

所以叠加的总数是边的总数， $m$ ，翻倍，因为我们考虑每条边两次。

因此，这个函数的增长顺序是  $o(n + m)$ ，这是一个方便的方法来说明运行时间按  $n$  或  $m$  的比例增长，无论哪个大。

如果我们知道  $n$  和  $m$  之间的关系，我们可以简化这个表达式。例如，在一个完全图中，边的个数为  $n(n-1)/2$ ，即  $o(n^2)$ 。所以对于一个完全图，可达节点在  $n$  中是二次的。

## 2.9 练习

本章的代码在 `chap02.ipynb` 中，这是本书资料库中的一个 Jupyter 笔记本。有关使用此代码的更多信息，请参见 0.3 节。

练习 2.1 启动 `chap02.ipynb` 并运行代码。笔记本里有一些简短的练习，你可以试试。

练习 2.2 在第 2.8 节中，我们分析了可达节点的性能，并将其分类为  $o(n + m)$ ，其中  $n$  是节点数， $m$  是边数。继续分析，增长的顺序是什么？

```
def is_connected(G):
    start = list(G)[0]
    reachable = reachable_nodes(G, start)
    return len(reachable) == len(G)
```

**Exercise 2.3** In my implementation of `reachable_nodes`, you might be bothered by the apparent inefficiency of adding *all* neighbors to the stack without checking whether they are already in `seen`. Write a version of this function that checks the neighbors before adding them to the stack. Does this “optimization” change the order of growth? Does it make the function faster?

**Exercise 2.4** There are actually two kinds of ER graphs. The one we generated in this chapter,  $G(n, p)$ , is characterized by two parameters, the number of nodes and the probability of an edge between nodes.

An alternative definition, denoted  $G(n, m)$ , is also characterized by two parameters: the number of nodes,  $n$ , and the number of edges,  $m$ . Under this definition, the number of edges is fixed, but their location is random.

Repeat the experiments we did in this chapter using this alternative definition. Here are a few suggestions for how to proceed:

1. Write a function called `m_pairs` that takes a list of nodes and the number of edges,  $m$ , and returns a random selection of  $m$  edges. A simple way to do that is to generate a list of all possible edges and use `random.sample`.
2. Write a function called `make_m_graph` that takes  $n$  and  $m$  and returns a random graph with  $n$  nodes and  $m$  edges.
3. Make a version of `prob_connected` that uses `make_m_graph` instead of `make_random_graph`.
4. Compute the probability of connectivity for a range of values of  $m$ .

How do the results of this experiment compare to the results using the first type of ER graph?

```
是 _connected (g) :  
    Start = list (g)[0]  
    可达 = 可达 _ 节点(g, 开始)  
    返回 len (reachable) == len (g)
```

练习 2.3 在我的可到达节点的实现中，你可能会被将所有邻居添加到堆栈中而不检查它们是否已经被看到的明显的线性所困扰。编写此函数的一个版本，在将邻居添加到堆栈之前检查它们。这种优化是否改变了增长的顺序？它是否使功能变得更快？

练习 2.4 实际上有两种 ER 图。我们在这一章中生成的， $g(n; p)$ ，是两个参数的拥有属性---- 节点数和节点间边的概率。

另一个拥有属性， $g(n; m)$ ，也是一个参数：节点数， $n$ ，和边的数量， $m$ 。在这个参数下，边的数量是固定的，但它们的位置是随机的。

用这个方法重复我们在本章中所做的实验。

以下是一些如何进行的建议：

1. 编写一个名为 `m_pairs` 的函数，该函数接受一个节点列表和边的个数  $m$ ，并返回随机选择的  $m$  条边。一个简单的方法是生成一个所有可能边的列表，并使用 `random.sample`。
2. 编写一个名为 `make_m_graph` 的函数，该函数接受  $n$  和  $m$ ，并返回一个具有  $n$  个节点和  $m$  条边的随机图。
3. 创建一个 `prob_connected` 版本，使用 `Make_m_graph` 代替制作随机图表。
4. 计算  $m$  值范围的连通概率。

这个实验的结果与使用第一类 ER 图的结果相比如何？

# Chapter 3

## Small World Graphs

Many networks in the real world, including social networks, have the “small world property”, which is that the average distance between nodes, measured in number of edges on the shortest path, is much smaller than expected.

In this chapter, I present Stanley Milgram’s famous Small World Experiment, which was the first demonstration of the small world property in a real social network. Then we’ll consider Watts-Strogatz graphs, which are intended as a model of small world graphs. I’ll replicate the experiment Watts and Strogatz performed and explain what it is intended to show.

Along the way, we’ll see two new graph algorithms: breadth-first search (BFS) and Dijkstra’s algorithm for computing the shortest path between nodes in a graph.

The code for this chapter is in `chap03.ipynb` in the repository for this book. More information about working with the code is in Section 0.3.

### 3.1 Stanley Milgram

Stanley Milgram was an American social psychologist who conducted two of the most famous experiments in social science, the Milgram experiment, which studied people’s obedience to authority (<http://thinkcomplex.com/>

## 第三章

### 小世界图表

现实世界中的许多网络，包括社交网络，都具有小世界特性”，即以最短路径上的边数量来衡量的节点间平均距离比预期的要小得多。

在本章中，我将介绍斯坦利·米尔格拉姆著名的小世界实验，这是第一次在真实的社交网络中演示小世界财产。然后，我们将考虑 Watts-Strogatz 图，它是用来作为小世界图的模型的。我将重复沃茨和斯特罗加茨的实验，并解释实验的目的。

在这个过程中，我们将看到两个新的图算法：宽度优先搜索(BFS)和 Dijkstra 算法，用于计算图中节点之间的最短路径。

本章的代码位于本书知识库中的 chap03.ipynb 中。

关于使用代码的更多信息请参见 0.3 部分。

#### 3.1 Stanley Milgram

美国社会心理学家 Stanley Milgram 进行了两个最著名的社会科学实验---- 米尔格拉姆实验，研究人们对权威的服从 <http://thinkcomplex.com/>

[milgram](#)) and the Small World Experiment, which studied the structure of social networks (<http://thinkcomplex.com/small>).

In the Small World Experiment, Milgram sent a package to several randomly-chosen people in Wichita, Kansas, with instructions asking them to forward an enclosed letter to a target person, identified by name and occupation, in Sharon, Massachusetts (which happens to be the town near Boston where I grew up). The subjects were told that they could mail the letter directly to the target person only if they knew him personally; otherwise they were instructed to send it, and the same instructions, to a relative or friend they thought would be more likely to know the target person.

Many of the letters were never delivered, but for the ones that were the average path length — the number of times the letters were forwarded — was about six. This result was taken to confirm previous observations (and speculations) that the typical distance between any two people in a social network is about “six degrees of separation”.

This conclusion is surprising because most people expect social networks to be localized — people tend to live near their friends — and in a graph with local connections, path lengths tend to increase in proportion to geographical distance. For example, most of my friends live nearby, so I would guess that the average distance between nodes in a social network is about 50 miles. Wichita is about 1600 miles from Boston, so if Milgram’s letters traversed typical links in the social network, they should have taken 32 hops, not 6.

## 3.2 Watts and Strogatz

In 1998 Duncan Watts and Steven Strogatz published a paper in *Nature*, “Collective dynamics of ‘small-world’ networks”, that proposed an explanation for the small world phenomenon. You can download it from <http://thinkcomplex.com/watts>.

Watts and Strogatz start with two kinds of graph that were well understood: random graphs and regular graphs. In a random graph, nodes are connected at random. In a regular graph, every node has the same number of neighbors. They consider two properties of these graphs, clustering and path length:

米尔格拉姆)和小世界实验, 这个实验研究的是社会网络的结构 (<http://thinkcomplex.com/Small>)。

在“小世界实验”(Small World Experiment)项目中, 米尔格拉姆向堪萨斯州威奇托市随机挑选的几个人发送了一个包裹, 要求他们将附件中的一封信寄给马萨诸塞州沙龙市的目标人物, 信中注明了目标人物的姓名和职业(这里恰好是我长大的波士顿附近的小镇)。受试者被告知, 只有当他们亲自认识目标人物时, 他们才能直接将信件寄给他们; 否则, 他们会被指示将信件和同样的指示寄给他们认为更有可能认识目标人物的亲戚或朋友。

许多信件从未被送达, 但是对于那些平均路径长度的信件 | 转发的次数 | 大约是 6 次。这个结果被用来反驳之前的观察(和推测), 即在一个社交网络中任何两个人之间的典型距离大约是 1 个六度分隔理论。

这个结论是令人惊讶的, 因为大多数人期望社交网络是本地化的 | 人们倾向于住在他们的朋友附近 | 并且在一个有本地联系的图表中, 路径长度倾向于按照地理距离的比例增加。例如, 我的大多数朋友都住在附近, 所以我猜社交网络中节点之间的平均距离大约是 50 英里。威奇托离波士顿大约 1600 英里, 所以如果米尔格拉姆的信件穿过了社交网络中的典型链接, 它们应该有 32 个跳数, 而不是 6 个。

### 3.2 瓦特和斯特罗加茨

1998 年, 邓肯·瓦茨和史蒂文·斯托加茨在《自然》杂志上发表了一篇论文《小世界网络的集体动力学》, 对小世界现象提出了一种解释。你可以从 <http://thinkcomplex.com/watts> 下载。

Watts 和 Strogatz 从两种很容易理解的图开始: 随机图和正则图。在一个随机图中, 节点是随机连通的。在一个正则图中, 每个节点具有相同数目的邻居。他们考虑了这些图的两个属性, 聚类和路径长度:

- Clustering is a measure of the “cliquishness” of the graph. In a graph, a **clique** is a subset of nodes that are all connected to each other; in a social network, a clique is a set of people who are all friends with each other. Watts and Strogatz defined a clustering coefficient that quantifies the likelihood that two nodes that are connected to the same node are also connected to each other.
- Path length is a measure of the average distance between two nodes, which corresponds to the degrees of separation in a social network.

Watts and Strogatz show that regular graphs have high clustering and high path lengths, whereas random graphs with the same size usually have low clustering and low path lengths. So neither of these is a good model of social networks, which combine high clustering with short path lengths.

Their goal was to create a **generative model** of a social network. A generative model tries to explain a phenomenon by modeling the process that builds or leads to the phenomenon. Watts and Strogatz proposed this process for building small-world graphs:

1. Start with a regular graph with  $n$  nodes and each node connected to  $k$  neighbors.
2. Choose a subset of the edges and “rewire” them by replacing them with random edges.

The probability that an edge is rewired is a parameter,  $p$ , that controls how random the graph is. With  $p = 0$ , the graph is regular; with  $p = 1$  it is completely random.

Watts and Strogatz found that small values of  $p$  yield graphs with high clustering, like a regular graph, and low path lengths, like a random graph.

In this chapter I replicate the Watts and Strogatz experiment in the following steps:

1. We’ll start by constructing a ring lattice, which is a kind of regular graph.
2. Then we’ll rewire it as Watts and Strogatz did.

聚类是对图中小集团的一种度量。在一个图中，一个小团体是所有相互连接的节点的子集；在一个社交网络中，一个小团体是一组彼此都是朋友的人。

Watts 和 Strogatz 建立了一个集群模型，证明了两个节点连接到同一个节点的可能性

也彼此相连。

路径长度是衡量两个节点之间的平均距离，它对应于社会网络中的分离程度。

Watts 和 Strogatz 表明，规则图具有较高的聚类性和路径长度，而具有相同大小的随机图通常具有较低的聚类性和路径长度。因此，这两者都不是一个好的社会网络模型，它结合了高聚类 and 短路径长度。

他们的目标是创建一个社交网络的生成模型。生成模型试图通过建模构建或导致某种现象的过程来解释这种现象。Watts 和 Strogatz 提出了构建小世界图表的过程：

1. 从一个有  $n$  个节点和每个节点连接到  $k$  个邻居的正则图开始。
2. 选择边的一个子集，用随机边替换它们。

边被重新布线的概率是一个参数， $p$ ，它控制图的随机程度。当  $p = 0$  时，图是正则的；当  $p = 1$  时，图是完全随机的。

Watts 和 Strogatz 发现，高聚类的  $p$  产量图的小值，如正则图，低路径长度，如随机图。

在这一章中，我按照以下步骤重复了 Watts 和 Strogatz 的实验：

1. 我们先来构造一个环格，它是一种正则图。
2. 然后我们会像瓦茨和斯特罗加茨那样重新接线。

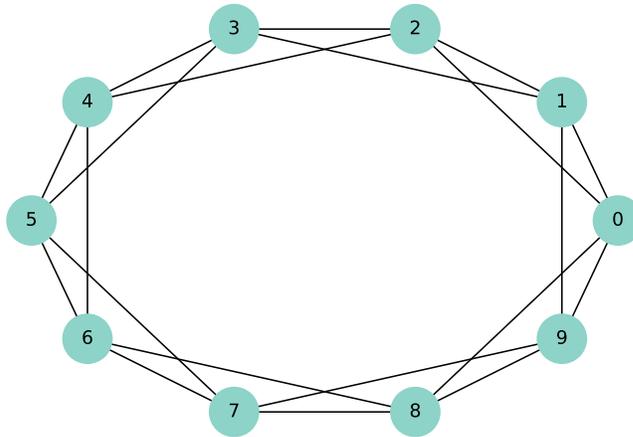


Figure 3.1: A ring lattice with  $n = 10$  and  $k = 4$ .

3. We'll write a function to measure the degree of clustering and use a NetworkX function to compute path lengths.
4. Then we'll compute the degree of clustering and path length for a range of values of  $p$ .
5. Finally, I'll present Dijkstra's algorithm, which computes shortest paths efficiently.

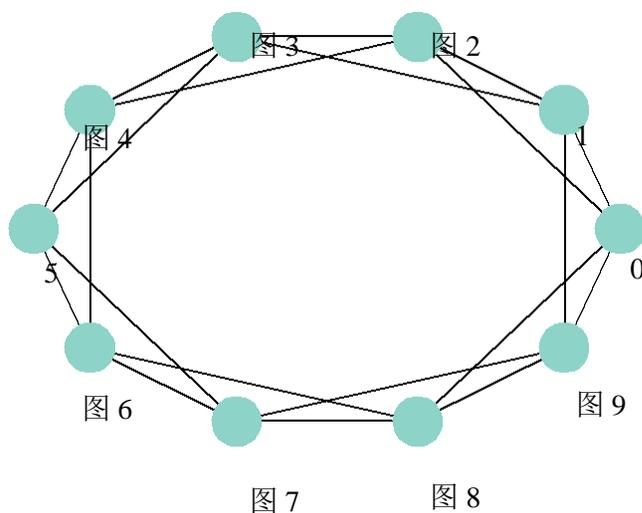
### 3.3 Ring lattice

A **regular** graph is a graph where each node has the same number of neighbors; the number of neighbors is also called the **degree** of the node.

A ring lattice is a kind of regular graph, which Watts and Strogatz use as the basis of their model. In a ring lattice with  $n$  nodes, the nodes can be arranged in a circle with each node connected to the  $k$  nearest neighbors.

For example, a ring lattice with  $n = 3$  and  $k = 2$  would contain the following edges:  $(0, 1)$ ,  $(1, 2)$ , and  $(2, 0)$ . Notice that the edges “wrap around” from the highest-numbered node back to 0.

More generally, we can enumerate the edges like this:

图 3.1:  $n = 10$  和  $k = 4$  的环点阵。

3. 我们将编写一个函数来度量聚类的程度，并使用 `NetworkX` 函数计算路径长度。
4. 然后我们计算  $p$  值范围的聚类度和路径长度。
5. 最后，我将介绍 `Dijkstra` 的算法，它可以精确地计算最短路径。

### 3.3 环形晶格

正则图是每个节点具有相同数量邻居的图，邻居的数量也称为节点的度数。

环格是一类正则图，`Watts` 和 `Strogatz` 用它作为模型的基础。在有  $n$  个节点的环格中，节点可以排成一个圆圈，每个节点与  $k$  个最近邻节点相连。

例如，具有  $n = 3$  和  $k = 2$  的环格包含以下边:  $(0; 1)$ 、 $(1; 2)$ 和 $(2; 0)$ 。请注意，“边”从编号最高的节点回绕到 0。

更一般地说，我们可以这样列举边缘:

```
def adjacent_edges(nodes, halfk):
    n = len(nodes)
    for i, u in enumerate(nodes):
        for j in range(i+1, i+halfk+1):
            v = nodes[j % n]
            yield u, v
```

`adjacent_edges` takes a list of nodes and a parameter, `halfk`, which is half of  $k$ . It is a generator function that yields one edge at a time. It uses the modulus operator, `%`, to wrap around from the highest-numbered node to the lowest.

We can test it like this:

```
>>> nodes = range(3)
>>> for edge in adjacent_edges(nodes, 1):
...     print(edge)
(0, 1)
(1, 2)
(2, 0)
```

Now we can use `adjacent_edges` to make a ring lattice:

```
def make_ring_lattice(n, k):
    G = nx.Graph()
    nodes = range(n)
    G.add_nodes_from(nodes)
    G.add_edges_from(adjacent_edges(nodes, k//2))
    return G
```

Notice that `make_ring_lattice` uses floor division to compute `halfk`, so it is only correct if  $k$  is even. If  $k$  is odd, floor division rounds down, so the result is a ring lattice with degree  $k-1$ . As one of the exercises at the end of the chapter, you will generate regular graphs with odd values of  $k$ .

We can test `make_ring_lattice` like this:

```
lattice = make_ring_lattice(10, 4)
```

Figure 3.1 shows the result.

```

Def 相邻的_棱(节点, 半 k):
    N = len(节点)
    对于 i, u in enumerate(nodes):
        对于 j in range(i + 1, i + halfk + 1):
            节点[j% n]
            放弃 u, v

```

相邻边接受一个节点列表和一个参数 `halfk`，它是 `k` 的一半。它是一个生成函数，每次生成一条边。它使用取模运算符 `%`，从编号最高的节点到编号最低的节点进行换行。

我们可以这样测试:

```

>>> 节点 = 范围(3)
>>> 对于相邻_边的边(节点, 1):
... 打印(边)(0,1)
(1,2)(2,
0)

```

现在我们可以使用相邻的边来构造一个环格:

```

Def make_ring_lattice(n, k):
    G = nx.Graph()
    节点 = 范围(n)
    G.add_nodes_from(nodes)
    G.add_edges_from(毗邻_边(节点, k//))
    返回 g

```

注意，`make_ring_lattice` 使用除法来计算 `halfk`，所以只有当 `k` 是偶数时才是正确的。如果 `k` 是奇数，则余除下，所以结果是一个 `k-1` 度的环格。作为本章最后的练习之一，您将生成奇数值为 `k` 的正则图。

我们可以这样测试环格:

```

晶格 = make_ring_lattice(10,4)

```

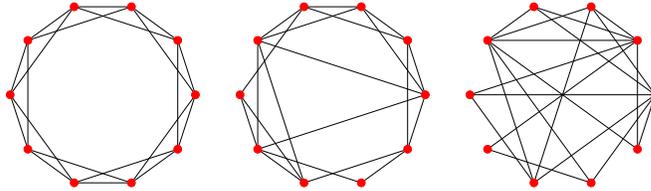


Figure 3.2: WS graphs with  $n = 20$ ,  $k = 4$ , and  $p = 0$  (left),  $p = 0.2$  (middle), and  $p = 1$  (right).

### 3.4 WS graphs

To make a Watts-Strogatz (WS) graph, we start with a ring lattice and “rewire” some of the edges. In their paper, Watts and Strogatz consider the edges in a particular order and rewire each one with probability  $p$ . If an edge is rewired, they leave the first node unchanged and choose the second node at random. They don’t allow self loops or multiple edges; that is, you can’t have an edge from a node to itself, and you can’t have more than one edge between the same two nodes.

Here is my implementation of this process.

```
def rewire(G, p):
    nodes = set(G)
    for u, v in G.edges():
        if flip(p):
            choices = nodes - {u} - set(G[u])
            new_v = np.random.choice(list(choices))
            G.remove_edge(u, v)
            G.add_edge(u, new_v)
```

The parameter  $p$  is the probability of rewiring an edge. The `for` loop enumerates the edges and uses `flip` (defined in Section 2.6) to choose which ones get rewired.

If we are rewiring an edge from node  $u$  to node  $v$ , we have to choose a replacement for  $v$ , called `new_v`.

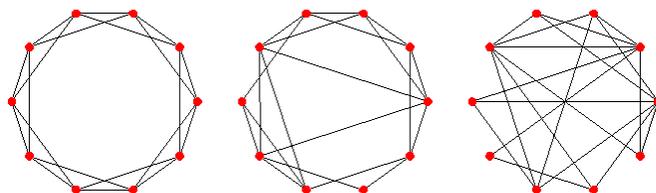


图 3.2:  $n = 20$ ,  $k = 4$ ,  $p = 0$ (左),  $p = 0.2$ (中),  $p = 1$ (右)的 WS 图。

### 3.4 WS 图表

为了构造 Watts-Strogatz (WS)图, 我们从一个环格开始, 重新连接“一些边”。在他们的论文中, Watts 和 Strogatz 以特定的顺序考虑边, 并以  $p$  的概率重新布线。如果一个边被重新连接, 它们保持第一个节点不变, 并随机选择第二个节点。它们不允许自循环或多个边; 也就是说, 你不能有一个从一个节点到它自己的边, 你不能有一个以上的边在同一两个节点之间。

下面是我对这个过程的实现。

```

Def rewire (g, p):
    节点 = set (g)
    对于 u, v in G.edges ():
        如果 flip (p):
            Choices = nodes - { u } - set (g [ u ])
            New _ v = np.random.choice (list (choices))
            删除边缘(u, v)
            (u, new _ v)

```

参数  $p$  是重新布线边缘的概率。For 循环枚举边缘并使用 flip (在 2.6 节中介绍)来选择哪些边缘需要重新布线。

如果我们重新连接从节点  $u$  到节点  $v$  的边缘, 我们必须选择  $v$  的替代品, 称为  $new\_v$ 。

1. To compute the possible choices, we start with `nodes`, which is a set, and subtract off `u` and its neighbors, which avoids self loops and multiple edges.
2. To choose `new_v`, we use the NumPy function `choice`, which is in the module `random`.
3. Then we remove the existing edge from `u` to `v`, and
4. Add a new edge from `u` to `new_v`.

As an aside, the expression `G[u]` returns a dictionary that contains the neighbors of `u` as keys. It is usually faster than using `G.neighbors` (see <http://thinkcomplex.com/neigh>).

This function does not consider the edges in the order specified by Watts and Strogatz, but that doesn't seem to affect the results.

Figure 3.2 shows WS graphs with  $n = 20$ ,  $k = 4$ , and a range of values of  $p$ . When  $p = 0$ , the graph is a ring lattice. When  $p = 1$ , it is completely random. As we'll see, the interesting things happen in between.

## 3.5 Clustering

The next step is to compute the clustering coefficient, which quantifies the tendency for the nodes to form cliques. A **clique** is a set of nodes that are completely connected; that is, there are edges between all pairs of nodes in the set.

Suppose a particular node,  $u$ , has  $k$  neighbors. If all of the neighbors are connected to each other, there would be  $k(k - 1)/2$  edges among them. The fraction of those edges that actually exist is the local clustering coefficient for  $u$ , denoted  $C_u$ .

If we compute the average of  $C_u$  over all nodes, we get the “network average clustering coefficient”, denoted  $\bar{C}$ .

Here is a function that computes it.

1. 为了计算可能的选择，我们从节点开始，这是一个集合，减去  $o_u$  和它的邻居，这样就避免了自循环和多边。
2. 要选择  $new\_v$ ，我们使用模块 `random` 中的 NumPy 函数 `choice`。
3. 然后我们将现有的边缘从  $u$  移除到  $v$ ，并且
4. 添加一个从  $u$  到  $new\ v$  的新边。

作为旁白，表达式 `g[u]` 返回一个包含  $u$  的邻居作为键的字典。它通常比使用 `G.neighbors` 更快(参见 [http:// thinkcomplex.com/neighbor](http://thinkcomplex.com/neighbor) 地图)。

这个函数没有按照 Watts 和 Strogatz 的顺序考虑边缘，但是这似乎不影响结果。

图 3.2 显示了  $n = 20$ ， $k = 4$  和  $p$  值范围的 WS 图。

当  $p = 0$  时，图是环格，当  $p = 1$  时，图是完全随机的。

正如我们将看到的，有趣的事情发生在这两者之间。

### 3.5 聚类

下一步是计算聚类中心，它反映了节点形成团体的倾向。团是一组完全连接的节点，即集合中所有节点对之间都有边。

假设一个特定的节点  $u$  有  $k$  个邻居。如果所有的邻居都相互连接，那么它们之间就有  $k(k-1)/2$  条边。实际存在的边的分数是  $u$  的局部聚类特征，表示  $c_u$ 。

如果我们计算所有节点的  $c_u$  的平均值，我们得到网络的平均值“聚类系数”，表示  $c$ 。

这是一个计算它的函数。

```

def node_clustering(G, u):
    neighbors = G[u]
    k = len(neighbors)
    if k < 2:
        return np.nan

    possible = k * (k-1) / 2
    exist = 0
    for v, w in all_pairs(neighbors):
        if G.has_edge(v, w):
            exist += 1
    return exist / possible

```

Again I use `G[u]`, which returns a dictionary with the neighbors of `u` as keys.

If a node has fewer than 2 neighbors, the clustering coefficient is undefined, so we return `np.nan`, which is a special value that indicates “Not a Number”.

Otherwise we compute the number of possible edges among the neighbors, count the number of those edges that actually exist, and return the fraction that exist.

We can test the function like this:

```

>>> lattice = make_ring_lattice(10, 4)
>>> node_clustering(lattice, 1)
0.5

```

In a ring lattice with  $k = 4$ , the clustering coefficient for each node is 0.5 (if you are not convinced, take another look at Figure 3.1).

Now we can compute the network average clustering coefficient like this:

```

def clustering_coefficient(G):
    cu = [node_clustering(G, node) for node in G]
    return np.nanmean(cu)

```

The NumPy function `nanmean` computes the mean of the local clustering coefficients, ignoring any values that are NaN.

We can test `clustering_coefficient` like this:

```
Defnode _ clustering (g, u):
```

```
    邻居 = g [ u ]
```

```
    K = len (邻居)
```

```
    如 k < 2:
```

```
        返回 np.nan
```

```
    可能 = k * (k-1)/2
```

```
    0
```

```
    对于 v, w 在所有 _对(邻居)中:
```

```
        如果 G.has _ edge (v, w):
```

```
            存在 += 1
```

```
    返回存在/可能
```

我再次使用 `g [ u ]`，它返回一个字典，`u` 的邻居作为键。

如果一个节点的邻居少于 2 个，则集群 coefficient 将被取消，因此我们返回 `np.nan`，这是一个特殊的值，表示“Not a Number”。

否则，我们计算邻居之间可能存在的边的数量，计算实际存在的边的数量，并返回存在的分数。

我们可以这样测试函数：

```
>>> 晶格 = make _ ring _ lattice (10,4)
```

```
>>> Node _ clustering (lattice, 1)
```

```
0.5
```

在  $k = 4$  的环格中，每个节点的集群顺序为 0.5(如果您不相信，请再看一下图 3.1)。

现在我们可以这样计算网络平均集群数据：

```
集群系数(g):
```

```
    Cu = [ node _ clustering (g, node) for node in g ]返回
```

```
    np.nanmean (cu)
```

NumPy 函数 `nanmean` 计算本地聚类参数的均值，忽略任何值为 NaN 的值。

```
>>> clustering_coefficient(lattice)
0.5
```

In this graph, the local clustering coefficient for all nodes is 0.5, so the average across nodes is 0.5. Of course, we expect this value to be different for WS graphs.

## 3.6 Shortest path lengths

The next step is to compute the characteristic path length,  $L$ , which is the average length of the shortest path between each pair of nodes. To compute it, I'll start with a function provided by NetworkX, `shortest_path_length`. I'll use it to replicate the Watts and Strogatz experiment, then I'll explain how it works.

Here's a function that takes a graph and returns a list of shortest path lengths, one for each pair of nodes.

```
def path_lengths(G):
    length_map = nx.shortest_path_length(G)
    lengths = [length_map[u][v] for u, v in all_pairs(G)]
    return lengths
```

The return value from `nx.shortest_path_length` is a dictionary of dictionaries. The outer dictionary maps from each node,  $u$ , to a dictionary that maps from each node,  $v$ , to the length of the shortest path from  $u$  to  $v$ .

With the list of lengths from `path_lengths`, we can compute  $L$  like this:

```
def characteristic_path_length(G):
    return np.mean(path_lengths(G))
```

And we can test it with a small ring lattice:

```
>>> lattice = make_ring_lattice(3, 2)
>>> characteristic_path_length(lattice)
1.0
```

In this example, all 3 nodes are connected to each other, so the mean path length is 1.

```
>>> 聚集系数(晶格)
0.5
```

在这个图中，所有节点的本地集群中心为 0.5，因此节点间的平均集群中心为 0.5。当然，我们希望这个值对于 WS 图是不同的。

### 3.6 最短路径长度

下一步是计算特征路径长度  $l$ ，它是每对节点之间最短路径的平均长度。要计算它，我将从 NetworkX 提供的函数开始，最短路径长度。我将用它来复制瓦茨和斯托加茨的实验，然后我将解释它是如何工作的。

这个函数接受一个图并返回一个最短路径长度的列表，每对节点一个。

返回的结果是:

```
长度_map = nx.shortpath_length(g)
长度 = [length_map[u][v] for u, v in all_pairs(g)] return length
```

来自 `nx.short_path_length` 的返回值是字典的字典。外部字典从每个节点  $u$  映射到一个字典，该字典从每个节点  $v$  映射到从  $u$  到  $v$  的最短路径长度。

通过路径长度的列表，我们可以这样计算  $l$ :

返回的结果是:

```
返回 np.mean(path_length(g))
```

我们可以用一个小的环形格子来测试它:

```
>>> 晶格 = make_ring_lattice(3,2)
>>> 特征路径长度(格子)
1.0
```

在这个例子中，所有 3 个节点相互连接，因此平均路径长度为 1。

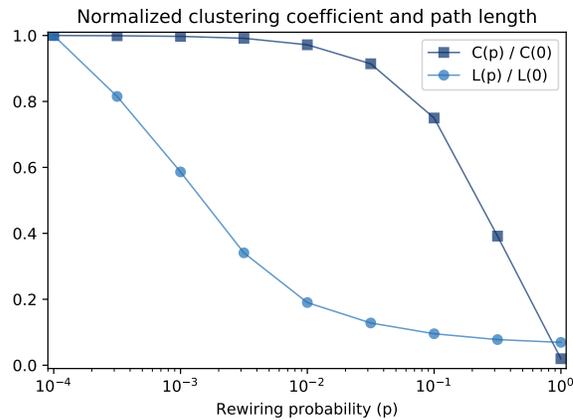


Figure 3.3: Clustering coefficient ( $C$ ) and characteristic path length ( $L$ ) for WS graphs with  $n = 1000$ ,  $k = 10$ , and a range of  $p$ .

### 3.7 The WS experiment

Now we are ready to replicate the WS experiment, which shows that for a range of values of  $p$ , a WS graph has high clustering like a regular graph and short path lengths like a random graph.

I'll start with `run_one_graph`, which takes  $n$ ,  $k$ , and  $p$ ; it generates a WS graph with the given parameters and computes the mean path length, `mpl`, and clustering coefficient, `cc`:

```
def run_one_graph(n, k, p):
    ws = make_ws_graph(n, k, p)
    mpl = characteristic_path_length(ws)
    cc = clustering_coefficient(ws)
    return mpl, cc
```

Watts and Strogatz ran their experiment with  $n=1000$  and  $k=10$ . With these parameters, `run_one_graph` takes a few seconds on my computer; most of that time is spent computing the mean path length.

Now we need to compute these values for a range of  $p$ . I'll use the NumPy function `logspace` again to compute `ps`:

```
ps = np.logspace(-4, 0, 9)
```

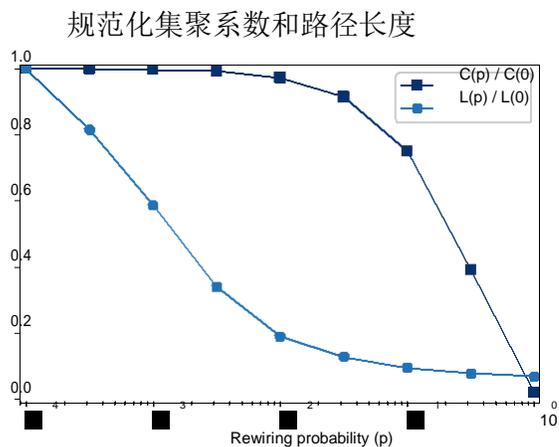


图 3.3:  $n = 1000$ ,  $k = 10$  和  $p$  范围的 WS 图的聚类 coefficient ( $c$ )和特征路径长度( $l$ )。

### 3.7 WS 实验

现在我们已经准备好重复 WS 实验，它表明对于  $p$  的一系列值，WS 图具有像正则图那样的高聚类性和像随机图那样的短路径长度。

我将从 `run one graph` 开始，它接受  $n$ ,  $k$  和  $p$ ; 它生成一个带有给定参数的 WS 图，并计算平均路径长度、`mpl` 和聚类 coefficient, `cc`:

```

1/graph (n k p) :
    ws = make_ws_graph (n, k, p)
    特征_path_length (ws)
    CC = clustering_coefficient (ws) return
    mpl, cc

```

和 Strogatz 用  $n = 1000$  和  $k = 10$  进行了实验。有了这些参数，在我的计算机上运行 `one_graph` 只需要几秒钟; 其中大部分时间用于计算平均路径长度。

现在我们需要计算  $p$  范围内的这些值，我将再次使用 NumPy 函数 `logspace` 来计算 `ps`:

```
Ps = np.logspace (-4,0,9)
```

Here's the function that runs the experiment:

```
def run_experiment(ps, n=1000, k=10, iters=20):
    res = []
    for p in ps:
        t = [run_one_graph(n, k, p) for _ in range(iters)]
        means = np.array(t).mean(axis=0)
        res.append(means)
    return np.array(res)
```

For each value of  $p$ , we generate 20 random graphs and average the results. Since the return value from `run_one_graph` is a pair, `t` is a list of pairs. When we convert it to an array, we get one row for each iteration and columns for  $L$  and  $C$ . Calling `mean` with the option `axis=0` computes the mean of each column; the result is an array with one row and two columns.

When the loop exits, `means` is a list of pairs, which we convert to a NumPy array with one row for each value of  $p$  and columns for  $L$  and  $C$ .

We can extract the columns like this:

```
L, C = np.transpose(res)
```

In order to plot  $L$  and  $C$  on the same axes, we standardize them by dividing through by the first element:

```
L /= L[0]
C /= C[0]
```

Figure 3.3 shows the results. As  $p$  increases, the mean path length drops quickly, because even a small number of randomly rewired edges provide shortcuts between regions of the graph that are far apart in the lattice. On the other hand, removing local links decreases the clustering coefficient much more slowly.

As a result, there is a wide range of  $p$  where a WS graph has the properties of a small world graph, high clustering and low path lengths.

And that's why Watts and Strogatz propose WS graphs as a model for real-world networks that exhibit the small world phenomenon.

下面是运行实验的函数:

```
Def run_experiment (ps, n = 1000, k = 10, iters = 20) :  
    Res = []  
    P 在 ps 里面:  
        T = [ run_one_graph (n, k, p) for _ in range (iters)] means =  
            np.array (t) . mean (axis = 0) res.append (means)  
    返回 np.array (res)
```

对于每个  $p$  值, 我们生成 20 个随机图并对结果进行平均。因为 `run_one_graph` 的返回值是一个对, 所以 `t` 是一个对的列表。当我们将其转换为数组时, 我们为每个迭代得到一行, 为 `l` 和 `c` 得到一列。

当循环退出时, 意味着是一个对的列表, 我们将其转换为一个 NumPy 数组, 其中每个值为一行, 列为 `l` 和 `c`。

我们可以这样提取列:

```
L, c = np.transpose (res)
```

为了将 `l` 和 `c` 绘制在同一个轴上, 我们通过除以第一个元素来标准化它们:

```
L/= l [0]  
C/= c [0]
```

图 3.3 显示了结果。随着  $p$  的增加, 平均路径长度迅速下降, 因为即使是一小部分随机重新布线的边, 也会在格子中相距较远的图的区域之间提供快捷方式。另一方面, 删除本地链接会使集群的可靠性降低得更慢。

因此, 在一个广泛的  $p$  范围内, WS 图具有小世界图、高聚类性和低路径长度的性质。

这就是为什么 Watts 和 Strogatz 提出 WS 图表作为展示小世界现象的现实世界网络的模型。

### 3.8 What kind of explanation is *that*?

If you ask me why planetary orbits are elliptical, I might start by modeling a planet and a star as point masses; I would look up the law of universal gravitation at <http://thinkcomplex.com/grav> and use it to write a differential equation for the motion of the planet. Then I would either derive the orbit equation or, more likely, look it up at <http://thinkcomplex.com/orbit>. With a little algebra, I could derive the conditions that yield an elliptical orbit. Then I would argue that the objects we consider planets satisfy these conditions.

People, or at least scientists, are generally satisfied with this kind of explanation. One of the reasons for its appeal is that the assumptions and approximations in the model seem reasonable. Planets and stars are not really point masses, but the distances between them are so big that their actual sizes are negligible. Planets in the same solar system can affect each other's orbits, but the effect is usually small. And we ignore relativistic effects, again on the assumption that they are small.

This explanation is also appealing because it is equation-based. We can express the orbit equation in a closed form, which means that we can compute orbits efficiently. It also means that we can derive general expressions for the orbital velocity, orbital period, and other quantities.

Finally, I think this kind of explanation is appealing because it has the form of a mathematical proof. It is important to remember that the proof pertains to the model and not the real world. That is, we can prove that an idealized model yields elliptical orbits, but we can't prove that real orbits are ellipses (in fact, they are not). Nevertheless, the resemblance to a proof is appealing.

By comparison, Watts and Strogatz's explanation of the small world phenomenon may seem less satisfying. First, the model is more abstract, which is to say less realistic. Second, the results are generated by simulation, not by mathematical analysis. Finally, the results seem less like a proof and more like an example.

Many of the models in this book are like the Watts and Strogatz model: abstract, simulation-based and (at least superficially) less formal than conventional mathematical models. One of the goals of this book is to consider the questions these models raise:

### 3.8 这是什么样的解释？

如果你问我为什么行星的轨道是椭圆的，我可能会从建模一个行星和一个恒星的点质量开始；我会在 <http://thinkcomplex.com/grav> 学院查找万有引力定律，并用它写出行星运动的直观方程。然后，我要么推导出轨道方程，或者更有可能的是，查阅 <http://thinkcomplex.com/orbit>。用一点代数，我就能推导出椭圆轨道的条件。然后我会说，我们认为行星上的物体满足这些条件。

人们，或者至少是科学家，通常对这种解释感到满意。其吸引力的原因之一是模型中的假设和近似似乎是合理的。行星和恒星实际上并不是点质量，但它们之间的距离太大，以至于它们的实际大小可以忽略不计。同一个太阳系的行星可以互相影响彼此的轨道，但影响通常很小。我们忽略相对论  $e$  效应，同样是基于它们很小的假设。

这种解释也很吸引人，因为它是基于方程式的。我们可以把轨道方程表示成一个封闭的形式，这意味着我们可以精确地计算轨道。这也意味着我们可以推导出轨道速度、轨道周期和其他量的一般表达式。

最后，我认为这种解释很有吸引力，因为它具有数学证明的形式。重要的是要记住，证明只适用于模型，而不适用于现实世界。也就是说，我们可以证明一个理想化的模型产生椭圆轨道，但我们不能证明真正的轨道是椭圆(事实上，它们不是)。尽管如此，证明的相似性还是很吸引人的。

相比之下，Watts 和 Strogatz 对小世界现象的解释似乎不那么令人满意。首先，这个模型比较抽象，也就是说比较不现实。其次，结果是通过模拟产生的，而不是数学分析。最后，结果似乎更像是一个例子，而不是一个证明。

本书中的许多模型都类似于 Watts 和 Strogatz 模型：与传统的数学模型相比，抽象的、基于模拟的和(至少是超常规的)不那么正式的。这本书的目的之一是考虑这些模型提出的问题：

- What kind of work can these models do: are they predictive, or explanatory, or both?
- Are the explanations these models offer less satisfying than explanations based on more traditional models? Why?
- How should we characterize the differences between these and more conventional models? Are they different in kind or only in degree?

Over the course of the book I will offer my answers to these questions, but they are tentative and sometimes speculative. I encourage you to consider them skeptically and reach your own conclusions.

## 3.9 Breadth-First Search

When we computed shortest paths, we used a function provided by NetworkX, but I have not explained how it works. To do that, I'll start with breadth-first search, which is the basis of Dijkstra's algorithm for computing shortest paths.

In Section 2.5 I presented `reachable_nodes`, which finds all the nodes that can be reached from a given starting node:

```
def reachable_nodes(G, start):
    seen = set()
    stack = [start]
    while stack:
        node = stack.pop()
        if node not in seen:
            seen.add(node)
            stack.extend(G.neighbors(node))
    return seen
```

I didn't say so at the time, but `reachable_nodes` performs a depth-first search (DFS). Now we'll modify it to perform breadth-first search (BFS).

To understand the difference, imagine you are exploring a castle. You start in a room with three doors marked A, B, and C. You open door C and discover another room, with doors marked D, E, and F.

这些模型能做什么工作: 它们是预测性的, 还是解释性的  
特瑞, 还是两者都是?

这些模型的解释是否不如基于传统模型的解释那么令人满意? 为什么?

我们应该如何描述这些模型和传统模型之间的差异? 它们是实物上的不同还是程度上的不同?

在本书中, 我将对这些问题做出回答, 但它们是试探性的, 有时是推测性的。我鼓励你以怀疑的眼光来看待它们, 并得出你自己的结论。

### 3.9 广度优先搜索

当我们计算最短路径时, 我们使用了 `NetworkX` 提供的一个函数, 但是我没有解释它是如何工作的。为了做到这一点, 我将首先从宽度——首次搜索开始, 它是 `Dijkstra` 计算最短路径的算法的基础。

在 2.5 节中, 我提出了可到达的节点, 它记录了所有可以从给定的起始节点到达的节点:

```
可到达的节点(g, 开始):
```

```
    看见 = set ()
```

```
    [开始]
```

```
    而 stack:
```

```
        Node = stack.pop ()
```

```
        如果节点没有出现:
```

```
            Add (node)
```

```
            Extend (G.neighbors (node))
```

```
    返回看到
```

当时我没有这么说, 但是可到达的节点执行深度优先搜索(DFS)。现在我们将修改它以执行宽度首次搜索(BFS)。

为了理解这种差异, 想象你正在探索一座城堡。你从一个有三扇门的房间开始, 门上分别标着 a, b 和 c。你打开 c 门, 发现另一个房间, 门上写着 d, e 和 f。

Which door do you open next? If you are feeling adventurous, you might want to go deeper into the castle and choose D, E, or F. That would be a depth-first search.

But if you wanted to be more systematic, you might go back and explore A and B before D, E, and F. That would be a breadth-first search.

In `reachable_nodes`, we use the list method `pop` to choose the next node to “explore”. By default, `pop` returns the last element of the list, which is the last one we added. In the example, that would be door F.

If we want to perform a BFS instead, the simplest solution is to pop the first element of the list:

```
node = stack.pop(0)
```

That works, but it is slow. In Python, popping the last element of a list takes constant time, but popping the first element is linear in the length of the list. In the worst case, the length of the stack is  $O(n)$ , which makes this implementation of BFS  $O(nm)$ , which is much worse than what it should be,  $O(n + m)$ .

We can solve this problem with a double-ended queue, also known as a **deque**. The important feature of a deque is that you can add and remove elements from the beginning or end in constant time. To see how it is implemented, see <http://thinkcomplex.com/deque>.

Python provides a deque in the `collections` module, so we can import it like this:

```
from collections import deque
```

And we can use it to write an efficient BFS:

```
def reachable_nodes_bfs(G, start):
    seen = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in seen:
            seen.add(node)
            queue.extend(G.neighbors(node))
    return seen
```

你接下来要打开哪扇门？如果你喜欢冒险，你可能想深入城堡，选择 d、e 或 f。这将是首次深度搜索。

但是如果你想变得更加系统化，你可以回顾一下，在 d、e 和 f 之前探索 a 和 b。这将是一次广泛的首次搜索。

在可到达的节点中，我们使用列表方法弹出选择下一个要探索的节点。默认情况下，`pop` 返回列表的最后一个元素，也是我们添加的最后一个元素。在这个例子中，门是 f。

如果我们想要执行 BFS，最简单的解决方案是弹出列表的第一个元素：

```
Node = stack.pop (0)
```

这是可行的，但是速度很慢。在 Python 中，弹出列表的最后一个元素需要常量时间，但是弹出第一个元素在列表长度中是线性的。在最坏的情况下，堆栈的长度是  $O(n)$ ，这使得 BFS  $O(nm)$  的实现比它应该的长度  $O(n + m)$  糟糕得多。

我们可以用双端队列来解决这个问题。Deque 的重要特性是，您可以添加和删除的元素从开始或结束在不变的时间。要了解它是如何实现的，请参阅 <http://thinkcomplex.com/deque>。

在 `collections` 模块中提供了 `deque`，所以我们可以这样导入它：

```
从托收进口货物
```

我们可以用它来写一个 efficient BFS：

```
可到达的 def _nodes_bfs (g, start):  
    看见 = set ()  
    Queue = deque ([ start ])    
    排队等候:  
        Node = queue.popleft ()  
        如果节点没有出现:  
            Add (node)  
            Queue.extend (G.neighbors (node))  
    返回看到
```

The differences are:

- I replaced the list called `stack` with a deque called `queue`.
- I replaced `pop` with `popleft`, which removes and returns the leftmost element of the queue.

This version is back to being  $O(n + m)$ . Now we're ready to find shortest paths.

## 3.10 Dijkstra's algorithm

Edsger W. Dijkstra was a Dutch computer scientist who invented an efficient shortest-path algorithm (see <http://thinkcomplex.com/dijk>). He also invented the semaphore, which is a data structure used to coordinate programs that communicate with each other (see <http://thinkcomplex.com/sem> and Downey, *The Little Book of Semaphores*).

Dijkstra is famous (and notorious) as the author of a series of essays on computer science. Some, like “A Case against the GO TO Statement”, had a profound effect on programming practice. Others, like “On the Cruelty of Really Teaching Computing Science”, are entertaining in their cantankerousness, but less effective.

**Dijkstra's algorithm** solves the “single source shortest path problem”, which means that it finds the minimum distance from a given “source” node to every other node in the graph (or at least every connected node).

I'll present a simplified version of the algorithm that considers all edges the same length. The more general version works with any non-negative edge lengths.

The simplified version is similar to the breadth-first search in the previous section except that we replace the set called `seen` with a dictionary called `dist`, which maps from each node to its distance from the source:

它们的区别是:

我用一个名为 `queue` 的 `deque` 替换了名为 `stack` 的列表。

我用 `popleft` 替换了 `pop`，它删除并返回队列的最左边的元素。

这个版本是回到  $O(n + m)$ 。现在我们准备好了第条最短的路径。

### 3.10 Dijkstra 的算法

艾兹赫尔·戴克斯特拉是一位荷兰计算机科学家，他发明了一种电子客户最短路径算法(见 <http://thinkcomplex.com/dijk>)。他还发明了信号量，这是一种数据结构，用于协调程序之间的通信(参见 <http://thinkcomplex.com/sem> 和唐尼的《信号量小书》)。

迪克斯特拉作为一系列计算机科学论文的作者而闻名(并且臭名昭著)。其中一些，比如《反对 `GO TO` 语句的案例》，对编程实践有着深远的影响。其他的，比如《论真正教授计算机科学的残酷性》，在他们的吵闹中很有趣，但是很少有反应。

Dijkstra 的算法解决了单源节点最短路径问题，这意味着它获得了从给定源节点到图中其他节点(或者至少是每个连接节点)的最小距离。

我将提出一个简化版本的算法，考虑到所有边的长度相同。更一般的版本工程与任何非负的边长度。

简化版本类似于前一节中的第一次广度搜索，只是我们用一个名为 `dist` 的字典替换了 `seen` 集，这个字典映射每个节点到它与源的距离:

```
def shortest_path_dijkstra(G, source):
    dist = {source: 0}
    queue = deque([source])
    while queue:
        node = queue.popleft()
        new_dist = dist[node] + 1

        neighbors = set(G[node]).difference(dist)
        for n in neighbors:
            dist[n] = new_dist

        queue.extend(neighbors)
    return dist
```

Here's how it works:

- Initially, `queue` contains a single element, `source`, and `dist` maps from `source` to distance 0 (which is the distance from `source` to itself).
- Each time through the loop, we use `popleft` to select the next node in the queue.
- Next we find all neighbors of `node` that are not already in `dist`.
- Since the distance from `source` to `node` is `dist[node]`, the distance to any of the undiscovered neighbors is `dist[node]+1`.
- For each neighbor, we add an entry to `dist`, then we add the neighbors to the queue.

This algorithm only works if we use BFS, not DFS. To see why, consider this:

1. The first time through the loop `node` is `source`, and `new_dist` is 1. So the neighbors of `source` get distance 1 and they go in the queue.
2. When we process the neighbors of `source`, all of *their* neighbors get distance 2. We know that none of them can have distance 1, because if they did, we would have discovered them during the first iteration.

返回文章页面最短路径译者:

```
0}
```

```
Queue = deque ([ source ])
```

排队等候:

```
Node = queue.popleft ()
```

```
New _ dist = dist [ node ] + 1
```

```
Neighbors = set (g [ node ] ) . difference (dist)
```

邻居中的 n:

新的, 新的, 新的

```
Queue.extend (neighbors)
```

返回距离

以下是它的工作原理:

最初, `queue` 包含单个元素、`source` 和 `dist` 映射源到距离 0(即从源到本身的距离)。

每次通过循环时, 我们使用 `popleft` 选择队列中的下一个节点。

接下来, 我们查找所有尚未在 `dist` 中的节点邻居。

因为从源到节点的距离是 `dist [ node ]`, 所以到任何未发现的邻居的距离是 `dist [ node ] + 1`。

对于每个邻居, 我们向 `dist` 添加一个条目, 然后将邻居添加到队列中。

这个算法只有在我们使用 `BFS` 而不是 `DFS` 的情况下才有效:

1. 循环节点中的第一次是 `source`, `new _ dist` 是 1。所以源的邻居得到距离 1, 他们排队。
2. 当我们处理源的邻居时, 所有邻居的距离都是 2。我们知道它们中没有一个可以有距离 1, 因为如果它们有, 我们会在第一次迭代中发现它们。

3. Similarly, when we process the nodes with distance 2, we give their neighbors distance 3. We know that none of them can have distance 1 or 2, because if they did, we would have discovered them during a previous iteration.

And so on. If you are familiar with proof by induction, you can see where this is going.

But this argument only works if we process all nodes with distance 1 before we start processing nodes with distance 2, and so on. And that's exactly what BFS does.

In the exercises at the end of this chapter, you'll write a version of Dijkstra's algorithm using DFS, so you'll have a chance to see what goes wrong.

## 3.11 Exercises

**Exercise 3.1** In a ring lattice, every node has the same number of neighbors. The number of neighbors is called the **degree** of the node, and a graph where all nodes have the same degree is called a **regular graph**.

All ring lattices are regular, but not all regular graphs are ring lattices. In particular, if  $k$  is odd, we can't construct a ring lattice, but we might be able to construct a regular graph.

Write a function called `make_regular_graph` that takes `n` and `k` and returns a regular graph that contains `n` nodes, where every node has `k` neighbors. If it's not possible to make a regular graph with the given values of `n` and `k`, the function should raise a `ValueError`.

**Exercise 3.2** My implementation of `reachable_nodes_bfs` is efficient in the sense that it is in  $O(n + m)$ , but it incurs a lot of overhead adding nodes to the queue and removing them. NetworkX provides a simple, fast implementation of BFS, available from the NetworkX repository on GitHub at <http://thinkcomplex.com/connx>.

Here is a version I modified to return a set of nodes:

3. 类似地，当我们处理距离为 2 的节点时，我们给它们的邻居距离为 3。我们知道它们中没有一个可以有距离 1 或 2，因为如果它们有，我们会在之前的迭代中发现它们。

等等，如果你熟悉数学归纳法，你就会明白这是怎么回事。

但是，只有在处理距离为 2 的节点之前，处理距离为 1 的所有节点时，这个参数才有效，依此类推。这正是 **BFS** 所做的。

在本章最后的练习中，你将使用 **DFS** 编写 **Dijkstra** 算法的一个版本，这样你就有机会看到哪里出错了。

### 3.11 练习

练习 3.1 在一个环形格子中，每个节点有相同数目的邻居。邻居的数目称为节点的度数，所有节点具有相同度数的图称为正则图。

所有的环格都是正则的，但并非所有的正则图都是环格。特别地，如果  $k$  是奇数，我们不能构造一个环格，但是我们可以构造一个正则图。

编写一个名为 `make_regular_graph` 的函数，该函数接受  $n$  和  $k$ ，并返回一个包含  $n$  个节点的正则图，其中每个节点都有  $k$  个邻居。如果不可能用给定的  $n$  和  $k$  值创建一个正则图，那么函数应该会产生一个 `ValueError`。

练习 3.2 我的可达节点 `_bfs` 的实现是在  $O(n + m)$  中实现的，但是将节点添加到队列并移除它们会带来很大的开销。提供了一个简单、快速的 **BFS** 实现，可以在 <http://thinkcomplex.com/connx> 的 GitHub 上的 **NetworkX** 存储库中找到。

下面是我修改后的返回一组节点版本:

```

def plain_bfs(G, start):
    seen = set()
    nextlevel = {start}
    while nextlevel:
        thislevel = nextlevel
        nextlevel = set()
        for v in thislevel:
            if v not in seen:
                seen.add(v)
                nextlevel.update(G[v])
    return seen

```

Compare this function to `reachable_nodes_bfs` and see which is faster. Then see if you can modify this function to implement a faster version of `shortest_path_dijkstra`.

**Exercise 3.3** The following implementation of BFS contains two performance errors. What are they? What is the actual order of growth for this algorithm?

```

def bfs(G, start):
    visited = set()
    queue = [start]
    while len(queue):
        curr_node = queue.pop(0)    # Dequeue
        visited.add(curr_node)

        # Enqueue non-visited and non-enqueued children
        queue.extend(c for c in G[curr_node]
                    if c not in visited and c not in queue)
    return visited

```

**Exercise 3.4** In Section 3.10, I claimed that Dijkstra’s algorithm does not work unless it uses BFS. Write a version of `shortest_path_dijkstra` that uses DFS and test it on a few examples to see what goes wrong.

**Exercise 3.5** A natural question about the Watts and Strogatz paper is whether the small world phenomenon is specific to their generative model or whether other similar models yield the same qualitative result (high clustering and low path lengths).



To answer this question, choose a variation of the Watts and Strogatz model and repeat the experiment. There are two kinds of variation you might consider:

- Instead of starting with a regular graph, start with another graph with high clustering. For example, you could put nodes at random locations in a 2-D space and connect each node to its nearest  $k$  neighbors.
- Experiment with different kinds of rewiring.

If a range of similar models yield similar behavior, we say that the results of the paper are **robust**.

**Exercise 3.6** Dijkstra’s algorithm solves the “single source shortest path” problem, but to compute the characteristic path length of a graph, we actually want to solve the “all pairs shortest path” problem.

Of course, one option is to run Dijkstra’s algorithm  $n$  times, once for each starting node. And for some applications, that’s probably good enough. But there are more efficient alternatives.

Find an algorithm for the all-pairs shortest path problem and implement it. See <http://thinkcomplex.com/short>.

Compare the run time of your implementation with running Dijkstra’s algorithm  $n$  times. Which algorithm is better in theory? Which is better in practice? Which one does NetworkX use?

为了回答这个问题，选择一个瓦茨和斯托加茨模型的变种，然后重复这个实验。你可以考虑两种变化：

与其从一个规则的图开始，不如从另一个高聚类的图开始。例如，可以将节点放在任意位置

在二维空间中，将每个节点连接到其最近的  $k$  个邻居。进行不同类型的重新布线试验。

如果一系列相似的模型产生了相似的行为，我们说这篇文章的结果是稳健的。

练习 3.6 Dijkstra 算法解决了单源最短路径问题，但是要计算图的特征路径长度，我们实际上是要解决所有对的最短路径问题。

当然，一种选择是运行 Dijkstra 的算法  $n$  次，每个起始节点运行一次。对于某些应用来说，这可能已经足够了。但是还有更多的电子替代品。

找到一个全对最短路径问题的算法并实现它。

参见 <http://thinkcomplex.com/short>。

将实现的运行时间与运行 Dijkstra 算法的运行时间比较  $n$  次。哪种算法在理论上更好？哪一个在实践中更好？NetworkX 使用哪一个？





# Chapter 4

## Scale-free networks

In this chapter, we'll work with data from an online social network, and use a Watts-Strogatz graph to model it. The WS model has characteristics of a small world network, like the data, but it has low variability in the number of neighbors from node to node, unlike the data.

This discrepancy is the motivation for a network model developed by Barabási and Albert. The BA model captures the observed variability in the number of neighbors, and it has one of the small world properties, short path lengths, but it does not have the high clustering of a small world network.

The chapter ends with a discussion of WS and BA graphs as explanatory models for small world networks.

The code for this chapter is in `chap04.ipynb` in the repository for this book. More information about working with the code is in Section 0.3.

### 4.1 Social network data

Watts-Strogatz graphs are intended to model networks in the natural and social sciences. In their original paper, Watts and Strogatz looked at the network of film actors (connected if they have appeared in a movie together); the electrical power grid in the western United States; and the network of neurons in the brain of the roundworm *C. elegans*. They found that all of

## 第四章

### 无标度网络

在本章中，我们将使用来自在线社交网络的数据，并使用 **Watts-Strogatz** 图表对其进行建模。**WS** 模型具有类似于数据的小世界网络的特点，但是与数据不同的是，它在节点间的邻居数量变化很小。

这种差异正是 **Barabasi** 和 **Albert** 开发网络模型的动机。**BA** 模型捕获了观察到的邻居数量的变化，它具有小世界特性之一，短路径长度，但它不具有小世界网络的高聚集性。

本章最后讨论了 **WS** 和 **BA** 图作为小世界网络的解释模型。

本章的代码见本书资料库第 04 章。

关于使用代码的更多信息请参见 0.3 部分。

#### 4.1 社交网络数据

**Watts-Strogatz** 图表旨在为自然科学和社会科学中的网络建模。在他们的原始论文中，**Watts** 和 **Strogatz** 研究了 **1m** 演员的网络(如果他们一起出现在电影中，就是连接在一起的)；美国西部的电力网络；以及秀丽蛔虫大脑中的神经网络。他们发现

these networks had the high connectivity and low path lengths characteristic of small world graphs.

In this section we'll perform the same analysis with a different dataset, a set of Facebook users and their friends. If you are not familiar with Facebook, users who are connected to each other are called “friends”, regardless of the nature of their relationship in the real world.

I'll use data from the Stanford Network Analysis Project (SNAP), which shares large datasets from online social networks and other sources. Specifically, I'll use their Facebook data<sup>1</sup>, which includes 4039 users and 88,234 friend relationships among them. This dataset is in the repository for this book, but it is also available from the SNAP website at <http://thinkcomplex.com/snap>.

The data file contains one line per edge, with users identified by integers from 0 to 4038. Here's the code that reads the file:

```
def read_graph(filename):
    G = nx.Graph()
    array = np.loadtxt(filename, dtype=int)
    G.add_edges_from(array)
    return G
```

NumPy provides a function called `loadtxt` that reads the given file and returns the contents as a NumPy array. The parameter `dtype` indicates that the “data type” of the array is `int`.

Then we use `add_edges_from` to iterate the rows of the array and make edges. Here are the results:

```
>>> fb = read_graph('facebook_combined.txt.gz')
>>> n = len(fb)
>>> m = len(fb.edges())
>>> n, m
(4039, 88234)
```

The node and edge counts are consistent with the documentation of the dataset.

---

<sup>1</sup>J. McAuley and J. Leskovec. Learning to Discover Social Circles in Ego Networks. NIPS, 2012.

这些网络具有小世界图所特有的高连通性和低路径长度。

在本节中，我们将对一个不同的数据集、一组 Facebook 用户和他们的朋友执行相同的分析。如果你不熟悉 Facebook，那么互相联系的用户就被称为朋友”，无论他们在现实世界中的关系如何。

我将使用来自斯坦福网络分析项目(SNAP)的数据，该项目共享来自在线社交网络和其他来源的大量数据集。特别地，我将使用他们的 Facebook 数据 1，其中包括 4039 个用户和 88234 个朋友关系。这个数据集存放在这本书的仓库里，但是也可以从 SNAP 网站上的 <http://thinkcomplex.com/SNAP> 图书馆获得。

数据 `le` 每条边包含一行，用户标识为 0 到 4038 之间的整数。下面是 `le` 的代码：

下面是一个例子：

```
G = nx. Graph ()
Array = np.loadtxt (filename, dtype = int)
从(数组)中添加 _ 棱角
返回 g
```

NumPy 提供了一个名为 `loadtxt` 的函数，该函数读取给定 `le` 并以 NumPy 数组的形式返回内容。参数 `dtype` 指示数组的数据类型为 `int`。

然后我们使用 `add _ edges _ from` 来迭代数组的行和创建边缘。

以下是调查结果：

```
>>> 如果你想知道更多关于 facebook 的信息，请访问我们的 facebook
>>> N = len (fb)
>>> M = len (fb.edges ())
>>> 好的，好的
(4039,88234)
```

节点和边缘计数与数据集的文档一致。

---

<sup>1</sup> 《学习在自我网络中发现社交圈子》，2012 年。

Now we can check whether this dataset has the characteristics of a small world graph: high clustering and low path lengths.

In Section 3.5 we wrote a function to compute the network average clustering coefficient. NetworkX provides a function called `average_clustering`, which does the same thing a little faster.

But for larger graphs, they are both too slow, taking time proportional to  $nk^2$ , where  $n$  is the number of nodes and  $k$  is the number of neighbors each node is connected to.

Fortunately, NetworkX provides a function that estimates the clustering coefficient by random sampling. You can invoke it like this:

```
from networkx.algorithms.approximation import average_clustering
average_clustering(G, trials=1000)
```

The following function does something similar for path lengths.

```
def sample_path_lengths(G, nodes=None, trials=1000):
    if nodes is None:
        nodes = list(G)
    else:
        nodes = list(nodes)

    pairs = np.random.choice(nodes, (trials, 2))
    lengths = [nx.shortest_path_length(G, *pair)
               for pair in pairs]
    return lengths
```

`G` is a graph, `nodes` is the list of nodes to sample from, and `trials` is the number of random paths to sample. If `nodes` is `None`, we sample from the entire graph.

`pairs` is a NumPy array of randomly chosen nodes with one row for each trial and two columns.

The list comprehension enumerates the rows in the array and computes the shortest distance between each pair of nodes. The result is a list of path lengths.

现在我们可以检查这个数据集是否具有小世界图的特征: 高聚类 and 低路径长度。

在 3.5 节中, 我们编写了一个函数来计算网络平均聚类数据。NetworkX 提供了一个名为 `average_clustering` 的函数, 它可以更快地完成同样的工作。

但是对于较大的图来说, 它们都太慢了, 花费的时间与  $nk^2$  成正比, 其中  $n$  是节点数,  $k$  是每个节点连接到的邻居数。

幸运的是, NetworkX 提供了一个通过随机抽样来估算聚类系数的函数, 你可以这样调用它:

```
来自 networkx.algorithm 近似导入平均值聚类平均值聚类(g, trials = 1000)
```

下面的函数对路径长度执行类似的操作。

```
函数的长度(g, nodes = None, trials = 1000):
```

```
    如果节点为 None:
```

```
        Nodes = list (g)
```

```
    其他:
```

```
        Nodes = list (nodes)
```

```
    Pairs = np.random.choice (节点, (试验, 2))
```

```
    长度 = [ nx.shortpath_length (g, * pair)
```

```
              [成双成对]
```

```
    回程长度
```

$G$  是一个图, 节点是要采样的节点列表, 试验是要采样的随机路径数。如果节点为 `None`, 则从整个图中抽样。

`Pairs` 是由随机选择的节点组成的 NumPy 数组, 每个试验有一行和两列。

列表内函数列举数组中的行并计算每对节点之间的最短距离。结果是一个路径长度列表。

`estimate_path_length` generates a list of random path lengths and returns their mean:

```
def estimate_path_length(G, nodes=None, trials=1000):
    return np.mean(sample_path_lengths(G, nodes, trials))
```

I'll use `average_clustering` to compute  $C$ :

```
C = average_clustering(fb)
```

And `estimate_path_lengths` to compute  $L$ :

```
L = estimate_path_lengths(fb)
```

The clustering coefficient is about 0.61, which is high, as we expect if this network has the small world property.

And the average path is 3.7, which is quite short in a network of more than 4000 users. It's a small world after all.

Now let's see if we can construct a WS graph that has the same characteristics as this network.

## 4.2 WS Model

In the Facebook dataset, the average number of edges per node is about 22. Since each edge is connected to two nodes, the average degree is twice the number of edges per node:

```
>>> k = int(round(2*m/n))
>>> k
44
```

We can make a WS graph with  $n=4039$  and  $k=44$ . When  $p=0$ , we get a ring lattice.

```
lattice = nx.watts_strogatz_graph(n, k, 0)
```

In this graph, clustering is high:  $C$  is 0.73, compared to 0.61 in the dataset. But  $L$  is 46, much higher than in the dataset!

估计路径长度生成一个随机路径长度列表，并返回它们的平均值：

```
Def estimate_path_length(g, nodes = None, trials = 1000):  
    返回 np.mean(sample_path_length(g, nodes, trials))
```

我将使用 `average_clustering` 来计算 `c`：

```
C = average_clustering(fb)
```

估计路径长度来计算 `l`：

```
L = estimate_path_length(fb)
```

群集的 `coe` 大约为 0.61，这是一个很高的值，正如我们所期望的，如果该网络具有小世界属性的话。

平均路径是 3.7，在一个超过 4000 个用户的网络中这是相当短的。毕竟这是一个小世界。

现在让我们看看是否可以构造一个具有与这个网络相同特征的 WS 图。

## 4.2 WS 模式

在 Facebook 数据集中，每个节点的平均边数约为 22 条。由于每个边连接到两个节点，平均度是每个节点边数的两倍：

```
>>> (2 * m/n)  
>>> K  
44
```

我们可以得到一个具有  $n = 4039$  和  $k = 44$  的 WS 图，当  $p = 0$  时，得到一个环格。

```
格点 = nx.watts_strogatz_graph(n, k, 0)
```

在这个图中，聚类是高的：`c` 是 0.73，而数据集是 0.61。但是 `l` 是 46，远远高于数据集！

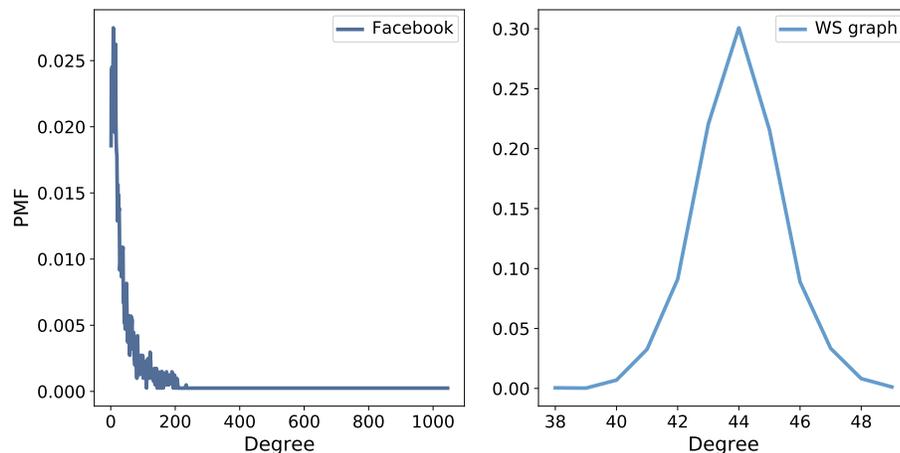


Figure 4.1: PMF of degree in the Facebook dataset and in the WS model.

With  $p=1$  we get a random graph:

```
random_graph = nx.watts_strogatz_graph(n, k, 1)
```

In the random graph,  $L$  is 2.6, even shorter than in the dataset (3.7), but  $C$  is only 0.011, so that's no good.

By trial and error, we find that when  $p=0.05$  we get a WS graph with high clustering and low path length:

```
ws = nx.watts_strogatz_graph(n, k, 0.05, seed=15)
```

In this graph  $C$  is 0.63, a bit higher than in the dataset, and  $L$  is 3.2, a bit lower than in the dataset. So this graph models the small world characteristics of the dataset well.

So far, so good.

## 4.3 Degree

If the WS graph is a good model for the Facebook network, it should have the same average degree across nodes, and ideally the same variance in degree.

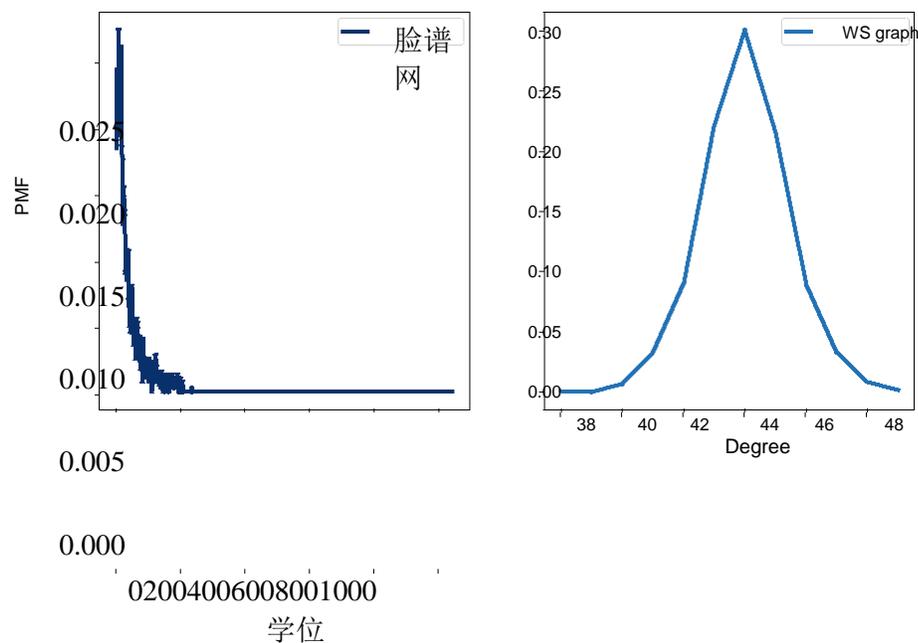


图 4.1: Facebook 数据集和 WS 模型中的 PMF。

对于  $p = 1$ ，我们得到一个随机图：

```
随机图 = nx.watts_strogatz_graph(n, k, 1)
```

在随机图中， $\langle l \rangle$  是 2.6，甚至比数据集(3.7)还要短，但是  $c$  只是 0.011，所以这不是很好。

通过反复试验，我们发现当  $p = 0.05$  时，我们得到了一个高聚类、低路径长度的 WS 图：

```
Ws = nx.watts_strogatz_graph(n, k, 0.05, seed = 15)
```

在这个图中  $c$  是 0.63，比数据集中高一点， $\langle l \rangle$  是 3.2，比数据集中低一点。所以这个图很好地模拟了数据集的小世界特征。

到目前为止，一切顺利。

This function returns a list of degrees in a graph, one for each node:

```
def degrees(G):  
    return [G.degree(u) for u in G]
```

The mean degree in model is 44, which is close to the mean degree in the dataset, 43.7.

However, the standard deviation of degree in the model is 1.5, which is not close to the standard deviation in the dataset, 52.4. Oops.

What's the problem? To get a better view, we have to look at the **distribution** of degrees, not just the mean and standard deviation.

I'll represent the distribution of degrees with a `Pmf` object, which is defined in the `thinkstats2` module. `Pmf` stands for “probability mass function”; if you are not familiar with this concept, you might want to read Chapter 3 of *Think Stats, 2nd edition* at <http://thinkcomplex.com/ts2>.

Briefly, a `Pmf` maps from values to their probabilities. A `Pmf` of degrees is a mapping from each possible degree,  $d$ , to the fraction of nodes with degree  $d$ .

As an example, I'll construct a graph with nodes 1, 2, and 3 connected to a central node, 0:

```
G = nx.Graph()  
G.add_edge(1, 0)  
G.add_edge(2, 0)  
G.add_edge(3, 0)  
nx.draw(G)
```

Here's the list of degrees in this graph:

```
>>> degrees(G)  
[3, 1, 1, 1]
```

Node 0 has degree 3, the others have degree 1. Now I can make a `Pmf` that represents this degree distribution:

```
>>> from thinkstats2 import Pmf  
>>> Pmf(degrees(G))  
Pmf({1: 0.75, 3: 0.25})
```

这个函数返回一个图中的度列表，每个节点一个：

```
Def degrees (g):  
    返回[ g 中 u 的度[ u ]
```

模型的平均度为 44，与数据集的平均度为 43.7 相近。

然而，模型中的学位标准差为 1.5，这与数据集中的标准差学位数据 52.4 不相符。哎呀。

有什么问题吗？为了得到一个更好的视图，我们必须观察学位的分布，而不仅仅是平均数和标准差。

我将使用 **Pmf** 对象表示度的分布，该对象是在 **thinkstats2** 模块中设计的。代表概率质量函数，如果你不熟悉这个概念，你可能想要阅读第三章的思考统计，第二版在 <http://thinkcomplex.com/ts2>。

布里，一个 **Pmf** 地图从价值到他们的概率。度的 **Pmf** 是从每个可能的度  $d$  到度  $d$  的节点分数的映射。

作为一个例子，我将构造一个图，其中 1、2 和 3 个节点连接到一个中心节点 0：

```
G = nx. Graph ()  
G.add _ edge (1,0)  
G.add _ edge (2,0)  
G.add _ edge (3,0)  
绘制(g)
```

下面是这张图表中的学位列表：

```
>>> 度(g)[3,1,1,1]
```

节点 0 有度 3，其他节点有度 1。现在我可以制作一个 **Pmf** 来代表这个学位分布：

```
>>> 2 import Pmf  
>>> Pmf (g)  
Pmf ({1:0.75,3:0.25})
```

The result is a `Pmf` object that maps from each degree to a fraction or probability. In this example, 75% of the nodes have degree 1 and 25% have degree 3.

Now we can make a `Pmf` that contains node degrees from the dataset, and compute the mean and standard deviation:

```
>>> from thinkstats2 import Pmf
>>> pmf_fb = Pmf(degrees(fb))
>>> pmf_fb.Mean(), pmf_fb.Std()
(43.691, 52.414)
```

And the same for the WS model:

```
>>> pmf_ws = Pmf(degrees(ws))
>>> pmf_ws.mean(), pmf_ws.std()
(44.000, 1.465)
```

We can use the `thinkplot` module to plot the results:

```
thinkplot.Pdf(pmf_fb, label='Facebook')
thinkplot.Pdf(pmf_ws, label='WS graph')
```

Figure 4.1 shows the two distributions. They are very different.

In the WS model, most users have about 44 friends; the minimum is 38 and the maximum is 50. That's not much variation. In the dataset, there are many users with only 1 or 2 friends, but one has more than 1000!

Distributions like this, with many small values and a few very large values, are called **heavy-tailed**.

## 4.4 Heavy-tailed distributions

Heavy-tailed distributions are a common feature in many areas of complexity science and they will be a recurring theme of this book.

We can get a clearer picture of a heavy-tailed distribution by plotting it on a log-log axis, as shown in Figure 4.2. This transformation emphasizes the tail of the distribution; that is, the probabilities of large values.

结果是一个 Pmf 对象，从每个度映射到一个分数或概率。在这个例子中，75% 的节点有度 1, 2.5% 有度 3。

现在我们可以从数据集中创建一个包含节点度数的 Pmf，并计算平均值和标准差：

```
>>> 2 import Pmf
>>> Pmf_fb = Pmf(度(fb))
>>> 平均值(), 平均值()
(43.691, 52.414)
```

WS 模式也是如此：

```
>>> Pmf_ws = Pmf(degrees(ws))
>>> 平均值(), 平均值()
(44.000, 1.465)
```

我们可以使用 thinkplot 模块绘制结果：

```
(pmf_fb, label = Facebook)
Thinkplot.Pdf(pmf_ws, label = 'WS graph')
```

图 4.1 显示了这两种分布，它们非常不同。

在 WS 模型中，大多数用户大约有 44 个好友，最小的是 38 个，最大的是 50 个。这种差异并不大。在数据集中，有许多用户只有 1 个或 2 个朋友，但是一个有超过 1000 个！

这样的分布，有许多小值和一些非常大的值，被称为重尾。

#### 4.4 重尾分布

重尾分布是复杂性科学许多领域的一个共同特征，它们将是本书反复出现的主题。

通过对数轴上绘制重尾分布，我们可以得到一个更清晰的图像，如图 4.2 所示。这种转换强调分布的尾部，也就是大值的概率。

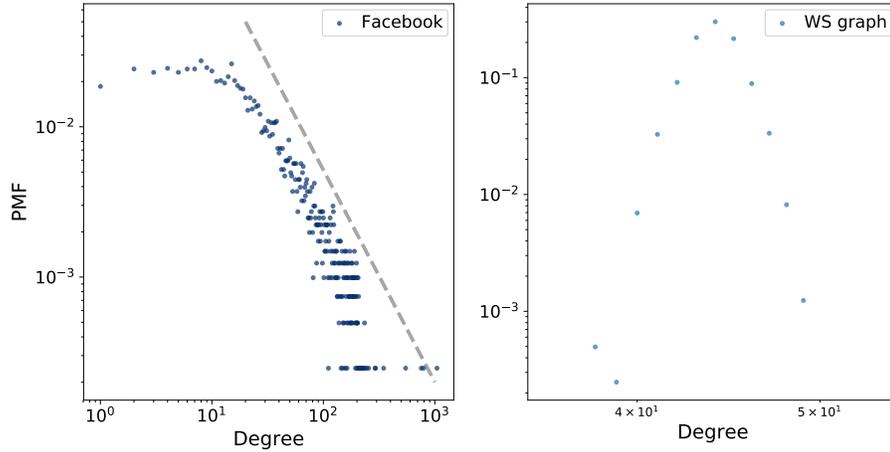


Figure 4.2: PMF of degree in the Facebook dataset and in the WS model, on a log-log scale.

Under this transformation, the data fall approximately on a straight line, which suggests that there is a **power law** relationship between the largest values in the distribution and their probabilities. Mathematically, a distribution obeys a power law if

$$\text{PMF}(k) \sim k^{-\alpha}$$

where  $\text{PMF}(k)$  is the fraction of nodes with degree  $k$ ,  $\alpha$  is a parameter, and the symbol  $\sim$  indicates that the PMF is asymptotic to  $k^{-\alpha}$  as  $k$  increases.

If we take the log of both sides, we get

$$\log \text{PMF}(k) \sim -\alpha \log k$$

So if a distribution follows a power law and we plot  $\text{PMF}(k)$  versus  $k$  on a log-log scale, we expect a straight line with slope  $-\alpha$ , at least for large values of  $k$ .

All power law distributions are heavy-tailed, but there are other heavy-tailed distributions that don't follow a power law. We will see more examples soon.

But first, we have a problem: the WS model has the high clustering and low path length we see in the data, but the degree distribution doesn't resemble

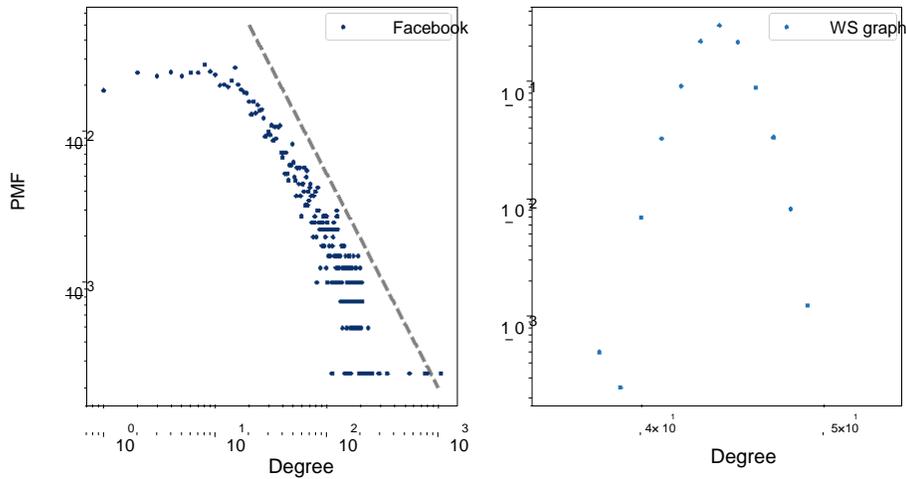


图 4.2: 脸书数据集和 WS 模型中的 PMF 值，以日志为尺度。

在这种变换下，数据近似下降到一条直线上，这表明分布中的最大值与其概率之间存在幂律关系。数学上，一个分布遵循幂定律，如果

$$PMF(k) \propto k^{-\alpha}$$

其中  $PMF(k)$  是  $k$  次节点的分数， $\alpha$  是一个参数，符号表明  $PMF$  随  $k$  的增加渐近于  $k^{-\alpha}$ 。

如果我们将两边的木头都取出来，我们就可以得到

$$\log PMF(k) = -\alpha \log k$$

因此，如果一个分布遵循幂律，我们在对数尺度上画  $PMF(k)$  和  $k$ ，我们期望一条斜率为  $-\alpha$  的直线，至少对于  $k$  的大值是这样。

所有的幂律分布都是重尾分布，但也有其他的重尾分布不遵循幂律。我们很快就会看到更多的例子。

但首先，我们有一个问题: WS 模型具有高的聚类性和低的路径长度，我们在数据中看到，但度分布不相似

the data at all. This discrepancy is the motivation for our next topic, the Barabási-Albert model.

## 4.5 Barabási-Albert model

In 1999 Barabási and Albert published a paper, “Emergence of Scaling in Random Networks”, that characterizes the structure of several real-world networks, including graphs that represent the interconnectivity of movie actors, web pages, and elements in the electrical power grid in the western United States. You can download the paper from <http://thinkcomplex.com/barabasi>.

They measure the degree of each node and compute  $PMF(k)$ , the probability that a vertex has degree  $k$ . Then they plot  $PMF(k)$  versus  $k$  on a log-log scale. The plots fit a straight line, at least for large values of  $k$ , so Barabási and Albert conclude that these distributions are heavy-tailed.

They also propose a model that generates graphs with the same property. The essential features of the model, which distinguish it from the WS model, are:

**Growth:** Instead of starting with a fixed number of vertices, the BA model starts with a small graph and adds vertices one at a time.

**Preferential attachment:** When a new edge is created, it is more likely to connect to a vertex that already has a large number of edges. This “rich get richer” effect is characteristic of the growth patterns of some real-world networks.

Finally, they show that graphs generated by the Barabási-Albert (BA) model have a degree distribution that obeys a power law.

Graphs with this property are sometimes called **scale-free networks**, for reasons I won’t explain; if you are curious, you can read more at <http://thinkcomplex.com/scale>.

NetworkX provides a function that generates BA graphs. We will use it first; then I’ll show you how it works.

```
ba = nx.barabasi_albert_graph(n=4039, k=22)
```

这种差异就是我们下一个话题——Barabasi-Albert 模型的动机。

#### 4.5 Barabasi-Albert 模型

1999 年, Barabasi 和 Albert 发表了一篇论文《随机网络中的伸缩现象》, 描述了几个真实世界网络的结构, 其中包括表示美国西部电影演员、网页和电力网络元素之间互连关系的图表。你可以从 <http://thinkcomplex.com/barabasi> 下载论文。

他们测量每个节点的度, 并计算 PMF ( $k$ ), 顶点有度  $k$  的概率。然后他们在对数尺度上绘制 PMF ( $k$ ) 和  $k$  的曲线。这些图是  $t$  直线, 至少对于  $k$  的大值, 所以 Barabasi 和 Albert 得出结论, 这些分布是重尾分布。

他们还提出了一个生成具有相同属性的图的模型。这个模式有别于 WS 模式的基本特点是:

增长: BA 模型不是以一定数量的顶点开始, 而是从一个小图开始, 一次增加一个顶点。

优先连接: 当创建一条新边时, 它更有可能连接到一个已经有大量边的顶点。这种富人越来越富有的方面是一些现实世界网络增长模式的特征。

最后, 他们证明了由 Barabasi-Albert (BA) 模型生成的图具有服从幂律的度分布。

具有这个属性的图有时被称为无标度网络, 原因我就不解释了; 如果你感到好奇, 可以在 [http:// thinkcomplex.com/scale](http://thinkcomplex.com/scale) 上阅读更多内容。

NetworkX 提供了一个生成 BA 图形的函数。我们将首先使用它; 然后我将向您展示它是如何工作的。

图( $n = 4039$ ,  $k = 22$ )

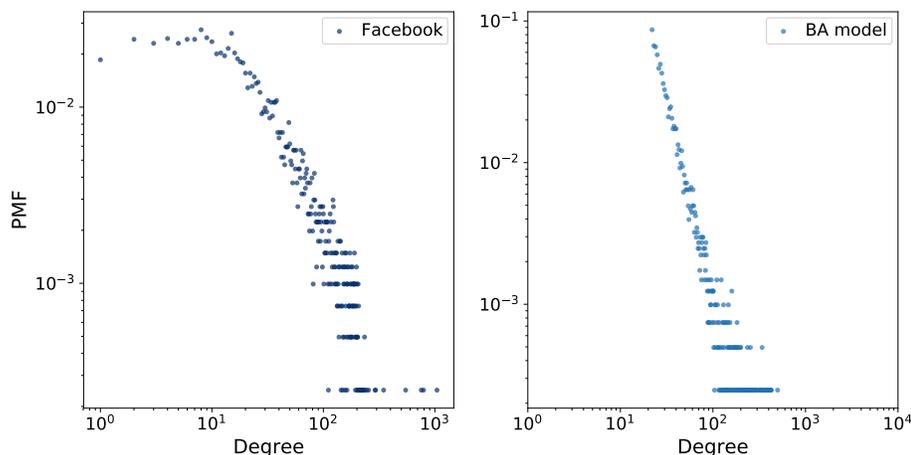


Figure 4.3: PMF of degree in the Facebook dataset and in the BA model, on a log-log scale.

The parameters are  $n$ , the number of nodes to generate, and  $k$ , the number of edges each node starts with when it is added to the graph. I chose  $k=22$  because that is the average number of edges per node in the dataset.

The resulting graph has 4039 nodes and 21.9 edges per node. Since every edge is connected to two nodes, the average degree is 43.8, very close to the average degree in the dataset, 43.7.

And the standard deviation of degree is 40.9, which is a bit less than in the dataset, 52.4, but it is much better than what we got from the WS graph, 1.5.

Figure 4.3 shows the degree distributions for the Facebook dataset and the BA model on a log-log scale. The model is not perfect; in particular, it deviates from the data when  $k$  is less than 10. But the tail looks like a straight line, which suggests that this process generates degree distributions that follow a power law.

So the BA model is better than the WS model at reproducing the degree distribution. But does it have the small world property?

In this example, the average path length,  $L$ , is 2.5, which is even more “small world” than the actual network, which has  $L = 3.69$ . So that’s good, although maybe too good.

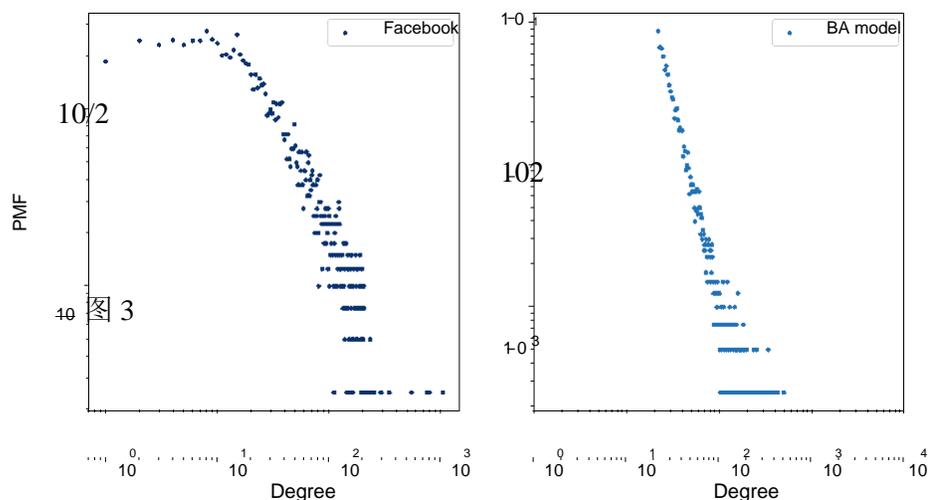


图 4.3: 脸谱网数据集和 BA 模型中的 PMF，在对数尺度上。

参数是  $n$ ，即要生成的节点数，以及  $k$ ，即每个节点添加到图中时的边数。我选择  $k = 22$  是因为这是数据集中每个节点的平均边数。

得到的图有 4039 个节点，每个节点有 21.9 条边。由于每个边都连接到两个节点，平均度为 43.8，非常接近数据集中的平均度为 43.7。

学位的标准差是 40.9，比数据集中的 52.4 要少一点，但是比我们从 WS 图表中得到的 1.5 要好得多。

图 4.3 显示了 Facebook 数据集和 BA 模型在日志尺度上的度分布。这个模型并不完美，特别是当  $k$  小于 10 时，它偏离了数据。但是尾巴看起来像一条直线，这表明这个过程产生了遵循幂定律的度分布。

因此，BA 模型在再现度分布方面优于 WS 模型。但是它有小世界的属性吗？

在这个示例中，平均路径长度  $l$  为 2.5，这比实际网络 ( $l = 3.69$ ) 更小。所以这很好，虽然可能太好了。

---

	Facebook	WS model	BA model
C	0.61	0.63	0.037
L	3.69	3.23	2.51
Mean degree	43.7	44	43.7
Std degree	52.4	1.5	40.1
Power law?	maybe	no	yes

---

Table 4.1: Characteristics of the Facebook dataset compared to two models.

On the other hand, the clustering coefficient,  $C$ , is 0.037, not even close to the value in the dataset, 0.61. So that's a problem.

Table 4.1 summarizes these results. The WS model captures the small world characteristics, but not the degree distribution. The BA model captures the degree distribution, at least approximately, and the average path length, but not the clustering coefficient.

In the exercises at the end of this chapter, you can explore other models intended to capture all of these characteristics.

## 4.6 Generating BA graphs

In the previous sections we used a `NetworkX` function to generate BA graphs; now let's see how it works. Here is a version of `barabasi_albert_graph`, with some changes I made to make it easier to read:

	Facebook	WS model	BA model
C	0.61	0.63	0.037
L	3.69	3.23	2.51
Mean degree	43.7	44	43.7
Std degree	52.4	1.5	40.1
Power law?	maybe	no	yes

表 4.1: 与两个模型相比 Facebook 数据集的特征。

另一方面，集群  $c$  为 0.037，甚至与数据集中的值，0.61。这是一个问题。

表 4.1 总结了这些结果。WS 模型捕获的是小世界特征，而不是度分布。BA 模型捕获度分布(至少是近似的)和平均路径长度，但不捕获集群特性。

在本章最后的练习中，您可以探索其他旨在捕捉所有这些特征的模型。

#### 4.6 生成 BA 图

在前面的部分中，我们使用了 `NetworkX` 函数来生成 BA 图；现在让我们看看它是如何工作的。下面是 `barabasi albert` 图表的一个版本，我做了一些改动使它更容易阅读：

```
def barabasi_albert_graph(n, k):  
  
    G = nx.empty_graph(k)  
    targets = list(range(k))  
    repeated_nodes = []  
  
    for source in range(k, n):  
        G.add_edges_from(zip([source]*k, targets))  
  
        repeated_nodes.extend(targets)  
        repeated_nodes.extend([source] * k)  
  
        targets = _random_subset(repeated_nodes, k)  
  
    return G
```

The parameters are  $n$ , the number of nodes we want, and  $k$ , the number of edges each new node gets (which will turn out to be the average number of edges per node).

We start with a graph that has  $k$  nodes and no edges. Then we initialize two variables:

The list of  $k$  nodes that will be connected to the next node. Initially `targets` contains the original  $k$  nodes; later it will contain a random subset of nodes.

A list of existing nodes where each node appears once for every edge it is connected to. When we select from `repeated_nodes`, the probability of selecting any node is proportional to the number of edges it has.

Each time through the loop, we add edges from the source to each node in `targets`. Then we update `repeated_nodes` by adding each target once and the new node  $k$  times.

Finally, we choose a subset of the nodes to be targets for the next iteration. Here's the definition of `_random_subset`:

```
(n, k):
```

```
G = nx.empty_graph(k)
Target = list(range(k))
重复的节点 = []
```

```
源在范围(k, n):
```

```
G.add_edges_from(zip([source] * k, targets))

重复_节点.扩展(目标)
重复_nodes.extend([source] * k)
```

```
目标 = _random_子集(重复_节点, k)
```

```
返回 g
```

参数是  $n$ ，即我们需要的节点数，以及  $k$ ，即每个新节点获得的边数(结果将是每个节点的平均边数)。

我们从一个有  $k$  个节点没有边的图开始，然后初始化两个变量：

将连接到下一个节点的  $k$  节点列表。最初的目标包含原始的  $k$  节点；稍后它将包含随机的节点子集。

现有节点的列表，其中每个节点对于所连接的每个边显示一次。当我们从重复的节点中进行选择时，选择任何节点的概率都与它所拥有的边的数量成正比。

每次通过循环，我们都将来自源的边添加到目标中的每个节点。然后通过添加每个目标一次和新节点  $k$  次来更新重复的节点。

最后，我们选择一个节点子集作为下一次迭代的目标。

下面是 `random` 子集的定义：

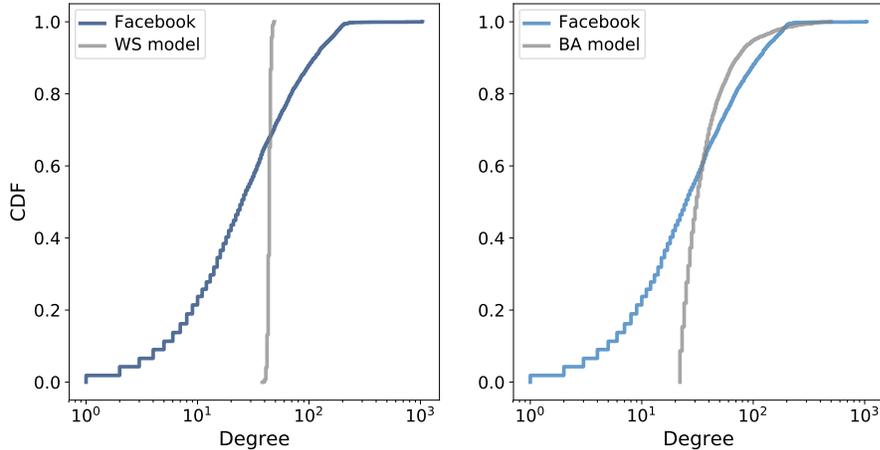


Figure 4.4: CDF of degree in the Facebook dataset along with the WS model (left) and the BA model (right), on a log-x scale.

```
def _random_subset(repeated_nodes, k):
    targets = set()
    while len(targets) < k:
        x = random.choice(repeated_nodes)
        targets.add(x)
    return targets
```

Each time through the loop, `_random_subset` chooses from `repeated_nodes` and adds the chosen node to `targets`. Because `targets` is a set, it automatically discards duplicates, so the loop only exits when we have selected `k` different nodes.

## 4.7 Cumulative distributions

Figure 4.3 represents the degree distribution by plotting the probability mass function (PMF) on a log-log scale. That’s how Barabási and Albert present their results and it is the representation used most often in articles about power law distributions. But it is not the best way to look at data like this.

A better alternative is a **cumulative distribution function** (CDF), which maps from a value,  $x$ , to the fraction of values less than or equal to  $x$ .

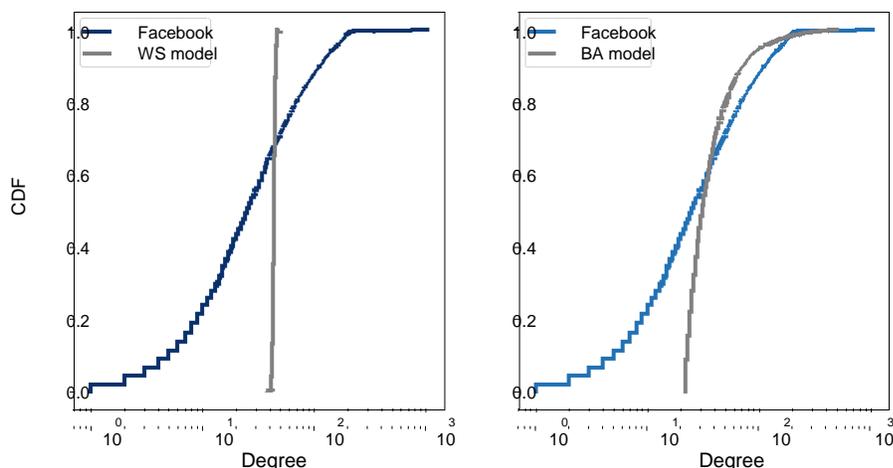


图 4.4: Facebook 数据集中度的 CDF, 以及在  $\log-x$  尺度上的 WS 模型(左)和 BA 模型(右)。

```
Def.random_subset (repeated_nodes, k):
```

```
    目标 = set ()
```

```
    而 len (target) < k:
```

```
        X = random.choice (repeated_nodes)
```

```
        目标 add (x)
```

```
    返回目标
```

每次通过循环, 随机子集从重复的节点中选择并将选择的节点添加到目标中。因为目标是一个集合, 所以它会自动丢弃重复的数据, 所以只有当我们选择了  $k$  个节点时, 循环才会退出。

#### 4.7 累积分布

图 4.3 通过在对数对数尺度上绘制概率质量函数分布(PMF)表示度分布。这就是 Barabasi 和 Albert 展示他们研究结果的方式, 也是在有关幂律分布的文章中最常用的方式。但这并不是看待这些数据的最佳方式。

一个更好的替代方法是累积分布函数映射(CDF), 它从一个值  $x$  映射到小于或等于  $x$  的值的分数。

Given a Pmf, the simplest way to compute a cumulative probability is to add up the probabilities for values up to and including  $x$ :

```
def cumulative_prob(pmf, x):
    ps = [pmf[value] for value in pmf if value<=x]
    return np.sum(ps)
```

For example, given the degree distribution in the dataset, `pmf_fb`, we can compute the fraction of users with 25 or fewer friends:

```
>>> cumulative_prob(pmf_fb, 25)
0.506
```

The result is close to 0.5, which means that the median number of friends is about 25.

CDFs are better for visualization because they are less noisy than PMFs. Once you get used to interpreting CDFs, they provide a clearer picture of the shape of a distribution than PMFs.

The `thinkstats` module provides a class called `Cdf` that represents a cumulative distribution function. We can use it to compute the CDF of degree in the dataset.

```
from thinkstats2 import Cdf
cdf_fb = Cdf(degrees(fb), label='Facebook')
```

And `thinkplot` provides a function called `Cdf` that plots cumulative distribution functions.

```
thinkplot.Cdf(cdf_fb)
```

Figure 4.4 shows the degree CDF for the Facebook dataset along with the WS model (left) and the BA model (right). The x-axis is on a log scale.

Clearly the CDF for the WS model is very different from the CDF from the data. The BA model is better, but still not very good, especially for small values.

In the tail of the distribution (values greater than 100) it looks like the BA model matches the dataset well enough, but it is hard to see. We can get

给定一个 Pmf，计算累积概率最简单的方法就是把值的概率加起来，最多等于或包括 x:

```
Def cumulative _ prob (pmf, x):  
    Ps = [ pmf [ value ] for value in pmf if value <= x ]返回  
    np.sum (ps)
```

例如，给定数据集 pmf fb 中的度分布，我们可以计算拥有 25 个或更少好友的用户比例:

```
>>> 累积 _ prob (pmf _ fb, 25)0.506
```

结果是接近 0.5，这意味着朋友数量的中位数约为 25。

因为 CDFs 比 PMFs 噪音小，所以可视化效果更好。一旦你习惯了解释 CDFs，他们提供了一个比 PMFs 更清晰的分布图形。

模块提供了一个名为 Cdf 的类，它表示一个累积分布函数。我们可以用它来计算数据集中度的 CDF。

```
2 import Cdf  
Cdf _ fb = Cdf (degrees (fb), label = ' Facebook')
```

并且 thinkplot 提供了一个名为 Cdf 的函数，用于绘制累积分布函数。

思想阴谋

图 4.4 显示了 Facebook 数据集的度 CDF 以及 WS 模型(左)和 BA 模型(右)。X 轴是对数刻度。

显然，WS 模型的 CDF 与来自数据的 CDF 是非常不同的。BA 模型更好，但仍然不是很好，特别是对于小值。

在分布的尾部(值大于 100)看起来 BA 模型与数据集足够匹配，但是很难看到。我们可以得到

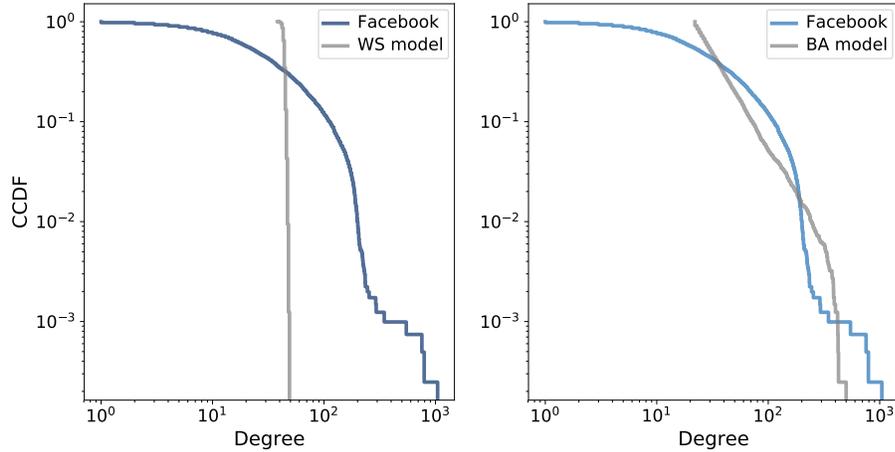


Figure 4.5: Complementary CDF of degree in the Facebook dataset along with the WS model (left) and the BA model (right), on a log-log scale.

a clearer view with one other view of the data: plotting the complementary CDF on a log-log scale.

The **complementary CDF** (CCDF) is defined

$$\text{CCDF}(x) \equiv 1 - \text{CDF}(x)$$

This definition is useful because if the PMF follows a power law, the CCDF also follows a power law:

$$\text{CCDF}(x) \sim \left( \frac{x}{x_m} \right)^{-\alpha}$$

where  $x_m$  is the minimum possible value and  $\alpha$  is a parameter that determines the shape of the distribution.

Taking the log of both sides yields:

$$\log \text{CCDF}(x) \sim -\alpha(\log x - \log x_m)$$

So if the distribution obeys a power law, we expect the CCDF on a log-log scale to be a straight line with slope  $-\alpha$ .

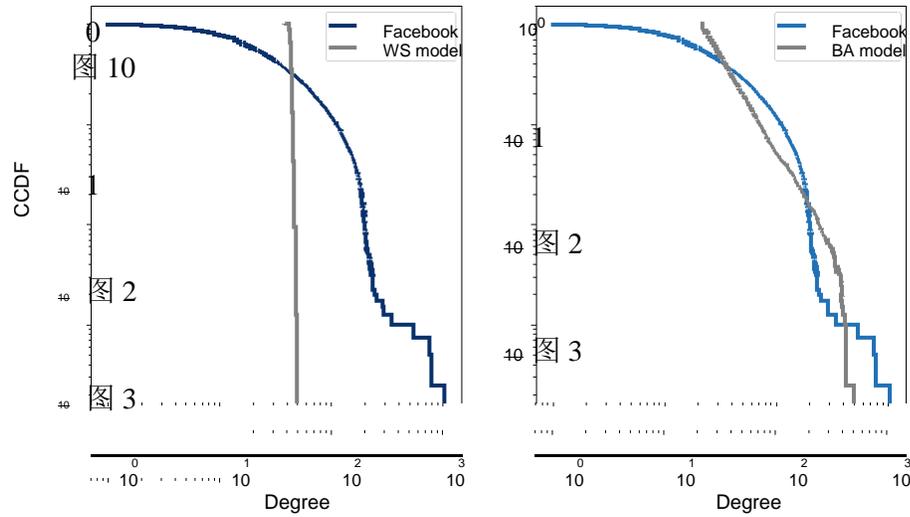


图 4.5: Facebook 数据集与 WS 模型(左)和 BA 模型(右)在对数尺度上的度互补 CDF。

一个更清晰的视图与其他视图的数据: 绘制互补的 CDF 在对数对数尺度。

设计了互补的 CDF (CCDF)

$$CCDF(x) = 1 - CDF(x)$$

这个定义很有用, 因为如果 PMF 遵循幂定律, 那么 CCDF 也遵循幂定律:

$$CCDF(x) \sim \frac{X}{X^m}$$

其中  $x_m$  是最小可能值, 并且是决定分布形状的参数。

把双方的木头都拿出来就知道了:

$$\log CCDF(x) \sim (\log x - \log x_m)$$

因此, 如果分布服从幂律, 我们期望对数尺度上的 CCDF 是一条斜率直线。

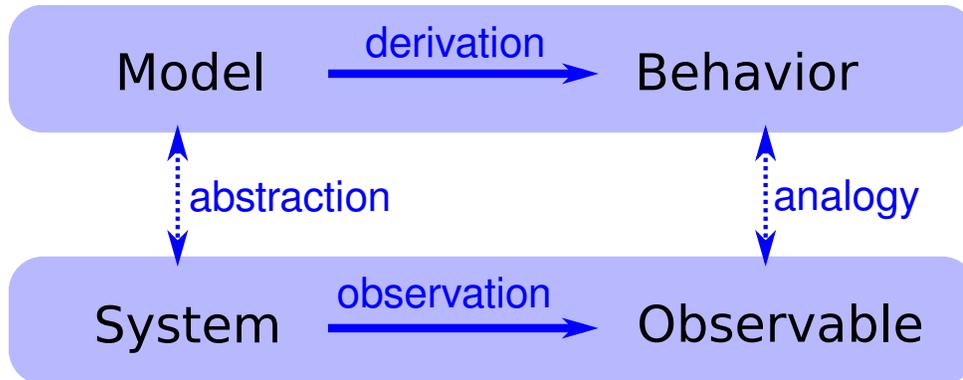


Figure 4.6: The logical structure of an explanatory model.

Figure 4.5 shows the CCDF of degree for the Facebook data, along with the WS model (left) and the BA model (right), on a log-log scale.

With this way of looking at the data, we can see that the BA model matches the tail of the distribution (values above 20) reasonably well. The WS model does not.

## 4.8 Explanatory models

We started the discussion of networks with Milgram’s Small World Experiment, which shows that path lengths in social networks are surprisingly small; hence, “six degrees of separation”.

When we see something surprising, it is natural to ask “Why?” but sometimes it’s not clear what kind of answer we are looking for. One kind of answer is an **explanatory model** (see Figure 4.6). The logical structure of an explanatory model is:

targeted in a system,  $S$ , we see something observable,  $O$ , that warrants explanation.

2. We construct a model,  $M$ , that is analogous to the system; that is, there is a correspondence between the elements of the model and the elements of the system.

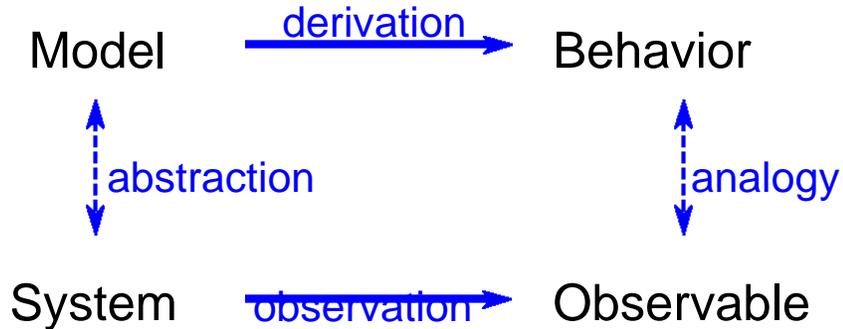


图 4.6: 解释性模型的逻辑结构。

图 4.5 显示了 Facebook 数据度的 CCDF，以及 WS 模型(左)和 BA 模型(右)。

通过这种查看数据的方式，我们可以看到 BA 模型与分布的尾部(值大于 20)匹配得相当好。而 WS 模式则不然。

#### 4.8 解释性模型

我们从米尔格拉姆的小世界实验开始讨论社交网络，这个实验表明社交网络中的路径长度小得惊人，因此我们称之为六度分隔理论。

当我们看到令人惊讶的东西时，自然会问为什么？”但有时候我们并不清楚，我们在寻找什么样的答案。一种答案是解释性模型(见图 4.6)。解释性模型的逻辑结构是：

目标: 重复 1. 在数据系统中，s，我们看到一些可以观察到的东西，o，这些东西值得解释。

2. 我们构造了一个模型，m，类似于系统；也就是说，在模型的元素和系统的元素之间有一个对应关系。

3. By simulation or mathematical derivation, we show that the model exhibits a behavior, B, that is analogous to O.
4. We conclude that S exhibits O *because* S is similar to M, M exhibits B, and B is similar to O.

At its core, this is an argument by analogy, which says that if two things are similar in some ways, they are likely to be similar in other ways.

Argument by analogy can be useful, and explanatory models can be satisfying, but they do not constitute a proof in the mathematical sense of the word.

Remember that all models leave out, or “abstract away”, details that we think are unimportant. For any system there are many possible models that include or ignore different features. And there might be models that exhibit different behaviors that are similar to O in different ways. In that case, which model explains O?

The small world phenomenon is an example: the Watts-Strogatz (WS) model and the Barabási-Albert (BA) model both exhibit elements of small world behavior, but they offer different explanations:

- The WS model suggests that social networks are “small” because they include both strongly-connected clusters and “weak ties” that connect clusters (see <http://thinkcomplex.com/weak>).
- The BA model suggests that social networks are small because they include nodes with high degree that act as hubs, and that hubs grow, over time, due to preferential attachment.

As is often the case in young areas of science, the problem is not that we have no explanations, but too many.

## 4.9 Exercises

**Exercise 4.1** In Section 4.8 we discussed two explanations for the small world phenomenon, “weak ties” and “hubs”. Are these explanations compatible; that is, can they both be right? Which do you find more satisfying as an explanation, and why?

3. 通过模拟或数学推导，我们证明了该模型具有类似于  $o$  的行为  $b$ 。
4. 我们的结论是  $s$  展品  $o$ ，因为  $s$  类似于  $m$ ， $m$  展品  $b$ ， $b$  类似于  $o$ 。

本质上，这是一个类推论证，即如果两件事情在某些方面相似，那么它们在其他方面也可能相似。

通过类比进行论证是有用的，解释性模型也是令人满意的，但它们不能构成词语数学意义上的证明。

请记住，所有的模型都会遗漏或抽象掉我们认为不重要的细节。对于任何系统，都有许多可能的模型包含或忽略不同的特性。而且可能有一些模型表现出与  $o$  在不同方面相似的不同行为。在这种情况下，哪种模式可以解释  $o$ ？

小世界现象就是一个例子: Watts-Strogatz (WS)模型和 Barabasi-Albert (BA)模型都展示了小世界行为的要素，但它们的解释却不同:

WS 模型表明，社交网络规模较小，“因为它们既包括强连接的集群，也包括弱连接”

集群(见 <http://thinkcomplex.com/weak>)。

英国广播公司模型表明，社交网络规模较小，因为它们包括具有高度枢纽作用的节点，而随着时间的推移，枢纽由于优先连接而不断增长。

正如在年轻的科学领域经常发生的那样，问题不是我们没有解释，而是解释太多了。

#### 4.9 练习

练习 4.1 在第 4.8 节中，我们讨论了小世界现象的两种解释: 弱关系和中心。这些解释是否一致，也就是说，它们都是正确的吗？你觉得哪种解释更令人满意，为什么？

Is there data you could collect, or experiments you could perform, that would provide evidence in favor of one model over the other?

Choosing among competing models is the topic of Thomas Kuhn’s essay, “Objectivity, Value Judgment, and Theory Choice”, which you can read at <http://thinkcomplex.com/kuhn>.

What criteria does Kuhn propose for choosing among competing models? Do these criteria influence your opinion about the WS and BA models? Are there other criteria you think should be considered?

**Exercise 4.2** NetworkX provides a function called `powerlaw_cluster_graph` that implements the “Holme and Kim algorithm for growing graphs with power-law degree distribution and approximate average clustering”. Read the documentation of this function (<http://thinkcomplex.com/hk>) and see if you can use it to generate a graph that has the same number of nodes as the Facebook dataset, the same average degree, and the same clustering coefficient. How does the degree distribution in the model compare to the actual distribution?

**Exercise 4.3** Data files from the Barabási and Albert paper are available from <http://thinkcomplex.com/netdata>. Their actor collaboration data is included in the repository for this book in a file named `actor.dat.gz`. The following function reads the file and builds the graph.

```
import gzip

def read_actor_network(filename, n=None):
    G = nx.Graph()
    with gzip.open(filename) as f:
        for i, line in enumerate(f):
            nodes = [int(x) for x in line.split()]
            G.add_edges_from(thinkcomplexity.all_pairs(nodes))
            if n and i >= n:
                break
    return G
```

Compute the number of actors in the graph and the average degree. Plot the PMF of degree on a log-log scale. Also plot the CDF of degree on a log-x

你是否可以收集一些数据，或者进行一些实验，这些数据可以提供证据支持一种模型胜过另一种模型？

在竞争模型中进行选择是 Thomas Kuhn 的文章《客观性、价值判断和理论选择》的主题，你可以在 <http://thinkcomplex.com/Kuhn> 读到这篇文章。

库恩建议在竞争模型中选择什么标准？这些标准是否影响了你对 WS 和 BA 模型的看法？还有其他你认为应该考虑的标准吗？

练习 4.2 NetworkX 提供了一个称为 `powerlaw_cluster_graph` 的函数，它实现了“关于幂律度分布和近似平均聚类的图的生长的 Holme 和 Kim 算法”。阅读这个函数 (<http://thinkcomplex.com/hk>) 的文档，看看是否可以使用它生成一个与 Facebook 数据集具有相同节点数、相同平均度和相同集群 coefficient 的图。模型中的程度分布与实际分布相比如何？

练习 4.3 来自 Barabasi 和 Albert 论文的数据来自 <http://thinkcomplex.com/netdata>。他们的 actor 协作数据包含在本书的存储库中，名为 `actor.dat.gz`。下面的函数读取 `le` 并构建图形。

输入 `gzip`

返回文章页面防御读演员\_网络(文件名, `n = None`):

```
G = nx.Graph()
```

```
使用 gzip.open(filename) 为 f:
```

```
    对于 i, 枚举(f)行:
```

```
        Nodes = [int(x) for x in line.split()]
```

```
        从(thinkcomplex.all_pair(nodes))中添加边
```

```
    如果 n 和 i >= n:
```

```
        打破
```

返回 `g`

计算图中参与者的数量和平均度。在对数尺度上绘制度的 PMF。还要在  $\log-x$  上绘制度的 CDF

scale, to see the general shape of the distribution, and on a log-log scale, to see whether the tail follows a power law.

Note: The actor network is not connected, so you might want to use `nx.connected_component_` to find connected subsets of the nodes.

---

用对数尺度来观察分布的总体形状，用对数尺度来观察尾巴是否遵循幂定律。

注意: 角色网络没有连接，因此您可能希望使用 `nx.connected_component_s` 到节点的 `nd connected` 子集。





# Chapter 5

## Cellular Automatons

A **cellular automaton** (CA) is a model of a world with very simple physics. “Cellular” means that the world is divided into discrete chunks, called cells. An “automaton” is a machine that performs computations — it could be a real machine, but more often the “machine” is a mathematical abstraction or a computer simulation.

This chapter presents experiments Stephen Wolfram performed in the 1980s, showing that some cellular automaton display surprisingly complicated behavior, including the ability to perform arbitrary computations.

I discuss implications of these results, and at the end of the chapter I suggest methods for implementing CAs efficiently in Python.

The code for this chapter is in `chap05.ipynb` in the repository for this book. More information about working with the code is in Section 0.3.

### 5.1 A simple CA

Cellular automaton<sup>1</sup> are governed by rules that determine how the state of the cells changes over time.

---

<sup>1</sup>You might also see the plural “automata”.

## 第五章

### 细胞自动机

细胞自动机是一个物理学非常简单的世界模型。“细胞”意味着世界被分割成离散的小块，称为细胞。自动机是一种执行计算的机器 | 它可能是一个真正的机器，但更多的时候，这种机器是一个数学抽象或计算机模拟。

本章介绍了 **Stephen Wolfram** 在 20 世纪 80 年代所做的实验，表明一些细胞自动机显示出惊人的复杂行为，包括执行任意计算的能力。

我讨论了这些结果的含义，并在本章的最后提出了在 `Python` 中实现 `ca` 的方法。

本章的代码位于本书知识库中的 `chap05.ipynb` 中。

关于使用代码的更多信息请参见 0.3 部分。

#### 5.1 一个简单的 CA

细胞自动机 1 受控于一些规则，这些规则决定细胞的状态如何随着时间的推移而变化。

---

<sup>1</sup> 你可能还会看到复数自动机”。

As a trivial example, consider a cellular automaton (CA) with a single cell. The state of the cell during time step  $i$  is an integer,  $x_i$ . As an initial condition, suppose  $x_0 = 0$ .

Now all we need is a rule. Arbitrarily, I'll pick  $x_{i+1} = x_i + 1$ , which says that during each time step, the state of the CA gets incremented by 1. So this CA performs a simple calculation: it counts.

But this CA is atypical; normally the number of possible states is finite. As an example, suppose a cell can only have one of two states, 0 or 1. For a 2-state CA, we could write a rule like  $x_{i+1} = (x_i + 1)\%2$ , where  $\%$  is the remainder (or modulus) operator.

The behavior of this CA is simple: it blinks. That is, the state of the cell switches between 0 and 1 during each time step.

Most CAs are **deterministic**, which means that rules do not have any random elements; given the same initial state, they always produce the same result. But some CAs are nondeterministic; we will see examples later.

The CA in this section has only one cell, so we can think of it as zero-dimensional. In the rest of this chapter, we explore one-dimensional (1-D) CAs; in the next chapter we explore two-dimensional CAs.

## 5.2 Wolfram's experiment

In the early 1980s Stephen Wolfram published a series of papers presenting a systematic study of 1-D CAs. He identified four categories of behavior, each more interesting than the last. You can read one of these papers, "Statistical mechanics of cellular automata," at <http://thinkcomplex.com/ca>.

In Wolfram's experiments, the cells are arranged in a lattice (which you might remember from Section 3.2) where each cell is connected to two neighbors. The lattice can be finite, infinite, or arranged in a ring.

The rules that determine how the system evolves in time are based on the notion of a "neighborhood", which is the set of cells that determines the next state of a given cell. Wolfram's experiments use a 3-cell neighborhood: the cell itself and its two neighbors.

作为一个简单的例子，考虑一个带有单个单元格的细胞自动机。单元格在时间步骤  $i$  中的状态是一个整数， $x_i$ 。作为初始条件，假设  $x_0 = 0$ 。

现在我们需要的是一个规则。任意地，我选择  $x_{i+1} = x_i + 1$ ，这意味着在每一个时间步骤中，CA 的状态会增加 1。所以这个 CA 执行一个简单的计算：它计数。

但是这个 CA 是非典型的；通常可能的状态数是零。例如，假设一个单元格只能有两种状态之一，0 或 1。对于一个 2 状态 CA，我们可以编写一个类似于  $x_{i+1} = (x_i + 1) \% 2$  的规则，其中  $\%$  是余数(或模数)运算符。

这个 CA 的行为很简单：它闪烁。也就是说，在每个时间步骤中，单元的状态在 0 和 1 之间切换。

大多数 ca 是确定性的，这意味着规则没有任何随机元素；给定相同的初始状态，它们总是产生相同的结果。但是有些 ca 是不确定的；我们将在后面看到示例。

本节中的 CA 只有一个单元格，因此我们可以将其视为零维的。在本章的其余部分中，我们将探讨一维(1-D) ca；在下一章中，我们将探讨二维 ca。

## 5.2 Wolfram 的实验

20 世纪 80 年代初，斯蒂芬·沃尔夫勒姆发表了一系列论文，对一维 ca 进行了系统的研究。他分析了四类行为，每一类都比前一类更有趣。你可以阅读这些论文中的一篇，《细胞自动机的统计力学》，在 <http://thinkcomplex.com/ca>。

在 Wolfram 的实验中，这些细胞排列在一个格子中(你可能记得 3.2 节)，每个细胞连接到两个相邻的细胞。晶格可以是黑色的，也可以是白色的，也可以是环状的。

决定系统在时间上如何演化的规则是基于“邻域”的概念，“邻域”是决定给定细胞下一个状态的一组细胞。Wolfram 的实验使用了一个 3 细胞邻居：细胞本身和它的两个邻居。

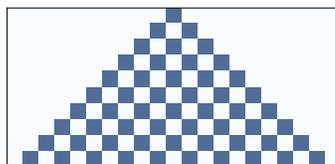


Figure 5.1: Rule 50 after 10 time steps.

In these experiments, the cells have two states, denoted 0 and 1 or “off” and “on”. A rule can be summarized by a table that maps from the state of the neighborhood (a tuple of three states) to the next state of the center cell. The following table shows an example:

prev	111	110	101	100	011	010	001	000
next	0	0	1	1	0	0	1	0

The first row shows the eight states a neighborhood can be in. The second row shows the state of the center cell during the next time step. As a concise encoding of this table, Wolfram suggested reading the bottom row as a binary number; because 00110010 in binary is 50 in decimal, Wolfram calls this CA “Rule 50”.

Figure 5.1 shows the effect of Rule 50 over 10 time steps. The first row shows the state of the system during the first time step; it starts with one cell “on” and the rest “off”. The second row shows the state of the system during the next time step, and so on.

The triangular shape in the figure is typical of these CAs; is it a consequence of the shape of the neighborhood. In one time step, each cell influences the state of one neighbor in either direction. During the next time step, that influence can propagate one more cell in each direction. So each cell in the past has a “triangle of influence” that includes all of the cells that can be affected by it.

## 5.3 Classifying CAs

How many of these CAs are there?

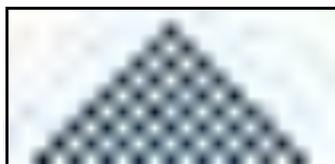


图 5.1:10 个时间步骤之后的规则 50。

在这些实验中，细胞有两种状态，表示 0 和 1 或 “o” 和 “on”。规则可以通过一个表进行汇总，该表将邻居的状态(三个状态的元组)映射到中心单元的下一个状态。下表列举了一个例子：

prev	111	110	101	100	011	010	001	000
next	0	0	1	1	0	0	1	0

第一行显示了一个社区可能处于的八个州。第二行显示下一个时间步骤期间中心单元格的状态。作为这个表格的一个简洁的编码，Wolfram 建议读取最下面一行作为一个二进制数；因为 00110010 在二进制中是 50 在十进制中，Wolfram 称之为 “CA Rule 50”。

图 5.1 显示了规则 50 在 10 个时间步骤中的影响。第一行显示第一个时间步骤期间的系统状态；它从“”上的一个单元格开始，其余的为“”。第二行显示下一个时间步骤期间系统的状态，依此类推。

图中的三角形是这些 ca 的典型特征，是邻里关系形成的结果。在一个时间步骤中，每个单元在任一方向上改变一个邻居的状态。在下一个时间步骤中，这个因子可以在每个方向上传播更多的细胞。所以过去的每个细胞都有一个因子三角形”，它包括所有能被它感应的细胞。

### 5.3 将核证机关分类

这些核证机关有多少个？

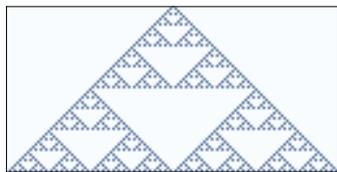


Figure 5.2: Rule 18 after 64 steps.

Since each cell is either on or off, we can specify the state of a cell with a single bit. In a neighborhood with three cells, there are 8 possible configurations, so there are 8 entries in the rule tables. And since each entry contains a single bit, we can specify a table using 8 bits. With 8 bits, we can specify 256 different rules.

One of Wolfram's first experiments with CAs was to test all 256 possibilities and classify them.

Examining the results visually, he proposed that the behavior of CAs can be grouped into four classes. Class 1 contains the simplest (and least interesting) CAs, the ones that evolve from almost any starting condition to the same uniform pattern. As a trivial example, Rule 0 always generates an empty pattern after one time step.

Rule 50 is an example of Class 2. It generates a simple pattern with nested structure, that is, a pattern that contains many smaller versions of itself. Rule 18 makes the nested structure is even clearer; Figure 5.2 shows what it looks like after 64 steps.

This pattern resembles the Sierpiński triangle, which you can read about at <http://thinkcomplex.com/sier>.

Some Class 2 CAs generate patterns that are intricate and pretty, but compared to Classes 3 and 4, they are relatively simple.

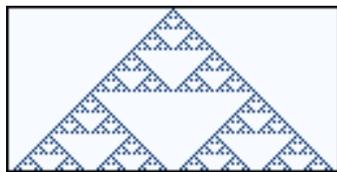


图 5.2:64 步之后的规则 18。

因为每个单元格都是 on 或 o，所以我们可以用一个位指定单元格的状态。在一个有三个单元格的邻居中，有 8 个可能的连接错误，所以规则表中有 8 个条目。由于每个条目包含一个位，我们可以使用 8 位指定一个表。用 8 位，我们可以指定 256 个不同的规则。

使用 ca 进行的第一个实验是测试所有 256 种可能性并对它们进行分类。

通过可视化地检查结果，他提出 ca 的行为可以分为四个类。类 1 包含最简单(也是最无趣的) ca，它们从几乎任何起始条件发展到相同的统一模式。作为一个简单的例子，Rule 0 总是在一个时间步骤之后生成一个空模式。

规则 50 是第二类的一个例子。它生成一个带有嵌套结构的简单模式，即一个包含许多自身的较小版本的模式。规则 18 使嵌套结构更加清晰; 图 5.2 显示了经过 64 个步骤后的结构。

这种模式类似于谢尔宾斯基三角形，你可以在《 <http://thinkcomplex.com/sier> 读到。

一些类 2 ca 生成的模式错综复杂，但与类 3 和类 4 相比，它们相对简单。

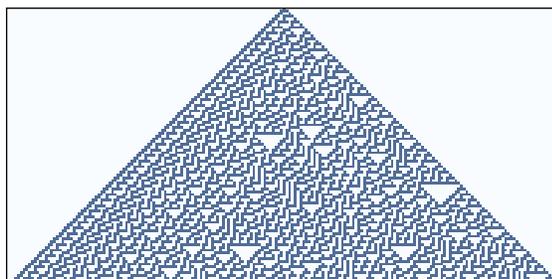


Figure 5.3: Rule 30 after 100 time steps.

## 5.4 Randomness

Class 3 contains CAs that generate randomness. Rule 30 is an example; Figure 5.3 shows what it looks like after 100 time steps.

Along the left side there is an apparent pattern, and on the right side there are triangles in various sizes, but the center seems quite random. In fact, if you take the center column and treat it as a sequence of bits, it is hard to distinguish from a truly random sequence. It passes many of the statistical tests people use to test whether a sequence of bits is random.

Programs that produce random-seeming numbers are called **pseudo-random number generators** (PRNGs). They are not considered truly random because:

- Many of them produce sequences with regularities that can be detected statistically. For example, the original implementation of `rand` in the C library used a linear congruential generator that yielded sequences with easily detectable serial correlations.
- Any PRNG that uses a finite amount of state (that is, storage) will eventually repeat itself. One of the characteristics of a generator is the **period** of this repetition.

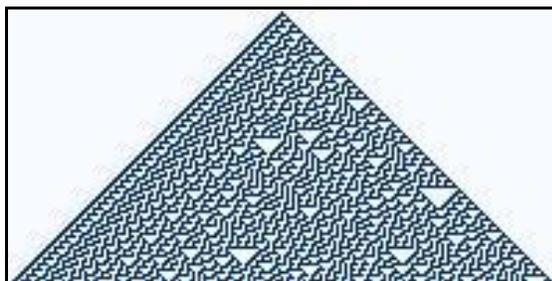


图 5.3:100 个时间步骤之后的规则 30。

#### 5.4 随机性

类 3 包含产生随机性的 `ca`。规则 30 就是一个例子; 图 5.3 显示了经过 100 个时间步骤后的结果。

沿着左边有一个明显的图案, 在右边有各种大小的三角形, 但中心似乎相当随机。事实上, 如果你把中间的列当作一个比特序列, 你很难区分它和一个真正的随机序列。它通过了人们用来测试一个比特序列是否是随机的许多统计测试。

产生表面上看似随机的数字的程序被称为伪随机数生成器(PRNGs)。它们不被认为是真正随机的, 因为:

它们中的许多产生的序列具有可以在统计上检测到的规律性。例如, `c` 库中 `rand` 的原始实现使用了一个可以生成序列的线性同余方法很容易发现的序列相关性。

任何使用 `nite` 量状态(即存储)的 PRNG 最终都会重复自己。生成器的特征之一是这种重复的周期。

- The underlying process is fundamentally deterministic, unlike some physical processes, like radioactive decay and thermal noise, that are considered to be fundamentally random.

Modern PRNGs produce sequences that are statistically indistinguishable from random, and they can be implemented with periods so long that the universe will collapse before they repeat. The existence of these generators raises the question of whether there is any real difference between a good quality pseudo-random sequence and a sequence generated by a “truly” random process. In *A New Kind of Science*, Wolfram argues that there is not (pages 315–326).

## 5.5 Determinism

The existence of Class 3 CAs is surprising. To explain how surprising, let me start with philosophical **determinism** (see <http://thinkcomplex.com/deter>). Many philosophical stances are hard to define precisely because they come in a variety of flavors. I often find it useful to define them with a list of statements ordered from weak to strong:

- D1:** Deterministic models can make accurate predictions for some physical systems.
- D2:** Many physical systems can be modeled by deterministic processes, but some are intrinsically random.
- D3:** All events are caused by prior events, but many physical systems are nevertheless fundamentally unpredictable.
- D4:** All events are caused by prior events, and can (at least in principle) be predicted.

My goal in constructing this range is to make D1 so weak that virtually everyone would accept it, D4 so strong that almost no one would accept it, with intermediate statements that some people accept.

The center of mass of world opinion swings along this range in response to historical developments and scientific discoveries. Prior to the scientific revolution, many people regarded the working of the universe as fundamentally

基本的过程是根本上确定的，不像一些物理过程，如放射性和热噪声，被认为是根本上随机的。

现代 PRNGs 产生的序列在统计学上与随机序列无法区分，而且它们的实现周期如此之长，以至于宇宙在它们重复之前就会崩溃。这些生成元的存在提出了一个问题，即一个高质量的伪随机序列和一个由真正的“随机过程”生成的序列之间是否存在真正的差异。在《一种新的科学》一书中，Wolfram 认为不存在这种现象(第 315 页 {326 页})。

### 5.5 决定论

第 3 类核证机关的存在令人惊讶。为了解释这是多么令人惊讶，让我从哲学决定论开始(参见 <http://thinkcomplex.com/> ·德特尔)。许多哲学立场很难精确地界定，因为它们有各种各样的立场。我经常觉得用一个从弱到强的语句列表来排列它们是有用的:

D1: 确定性模型可以对某些物理系统做出准确的预测。

D2: 许多物理系统可以用确定性过程来建模，但有些系统本质上是随机的。

D3: 所有事件都是由先前的事件引起的，但是许多物理系统从根本上来说是不可预测的。

D4: 所有的事件都是由先前的事件引起的，并且(至少在原则上)是可以预测的。

我构造这个范围的目标是使 d 1 变得如此弱，以至于几乎每个人都会接受它，d 4 变得如此强，以至于几乎没有人会接受它，中间的陈述有些人会接受。

世界舆论的中心随着历史发展和科学发现而在这个范围内摇摆。在科学革命之前，许多人认为宇宙的运行是根本性的

unpredictable or controlled by supernatural forces. After the triumphs of Newtonian mechanics, some optimists came to believe something like D4; for example, in 1814 Pierre-Simon Laplace wrote:

We may regard the present state of the universe as the effect of its past and the cause of its future. An intellect which at a certain moment would know all forces that set nature in motion, and all positions of all items of which nature is composed, if this intellect were also vast enough to submit these data to analysis, it would embrace in a single formula the movements of the greatest bodies of the universe and those of the tiniest atom; for such an intellect nothing would be uncertain and the future just like the past would be present before its eyes.

This “intellect” is now called “Laplace’s Demon”. See <http://thinkcomplex.com/demon>. The word “demon” in this context has the sense of “spirit”, with no implication of evil.

Discoveries in the 19th and 20th centuries gradually dismantled Laplace’s hope. Thermodynamics, radioactivity, and quantum mechanics posed successive challenges to strong forms of determinism.

In the 1960s chaos theory showed that in some deterministic systems prediction is only possible over short time scales, limited by precision in the measurement of initial conditions.

Most of these systems are continuous in space (if not time) and nonlinear, so the complexity of their behavior is not entirely surprising. Wolfram’s demonstration of complex behavior in simple cellular automata is more surprising — and disturbing, at least to a deterministic world view.

So far I have focused on scientific challenges to determinism, but the longest-standing objection is the apparent conflict between determinism and human free will. Complexity science provides a possible resolution of this conflict; I’ll come back to this topic in Section 10.6.

## 5.6 Spaceships

The behavior of Class 4 CAs is even more surprising. Several 1-D CAs, most notably Rule 110, are **Turing complete**, which means that they can compute

不可预测的或者受超自然力量控制的。在牛顿运动定律的胜利之后，一些乐观主义者开始相信类似 d4 的东西；例如，在 1814 年，皮埃尔-西蒙 拉普拉斯写道：

我们可以把宇宙的现状看作是它的过去和未来的原因。在某一时刻，一个知识分子将会知道所有引起自然运动的力量，以及所有构成自然的物体的所有位置，如果这个知识分子也足够广博，能够将这些数据提交给分析，它将会在一个单一的公式中包含宇宙中最大的物体和最小的原子的运动；因为这样一个知识分子没有什么是不确定的，未来就像过去一样呈现在它的眼前。

这种智力现在被称为拉普拉斯之魔。参见 <http://thinkcomplex.com/demon>。“恶魔”这个词在这个语境中具有精神意义，没有邪恶的含义。

19 世纪和 20 世纪的发现逐渐瓦解了拉普拉斯的希望。热力学、放射性和量子力学对决定论的强烈形式提出了连续的挑战。

20 世纪 60 年代的混沌理论表明，在某些确定性系统中，预测只能在短时间尺度上进行，这受到初始条件测量精度的限制。

这些系统中的大多数在空间(如果不是时间)是连续的和非线性的，所以它们行为的复杂性并不完全令人惊讶。沃尔夫勒姆在简单的细胞自动机中展示的复杂行为更令人惊讶和不安，至少对于一个确定性的世界观来说是这样的。

到目前为止，我主要关注的是对决定论的科学挑战，但最长久的异议是决定论与人类自由意志之间的明显冲突。复杂性科学为这种冲突提供了一种可能的解决方案；我将在第 10.6 节回到这个主题。

## 5.6 宇宙飞船

类 4 ca 的行为更令人惊讶。几个一维 ca (最著名的是 Rule 110)都是图灵完备的，这意味着它们可以进行计算

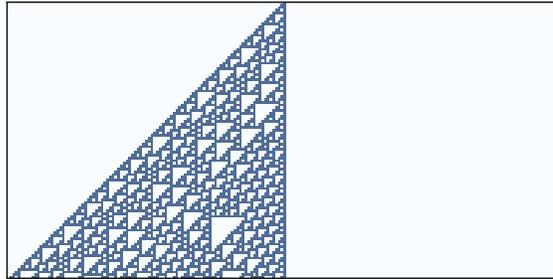


Figure 5.4: Rule 110 after 100 time steps.

any computable function. This property, also called **universality**, was proved by Matthew Cook in 1998. See <http://thinkcomplex.com/r110>.

Figure 5.4 shows what Rule 110 looks like with an initial condition of a single cell and 100 time steps. At this time scale it is not apparent that anything special is going on. There are some regular patterns but also some features that are hard to characterize.

Figure 5.5 shows a bigger picture, starting with a random initial condition and 600 time steps:

After about 100 steps the background settles into a simple repeating pattern, but there are a number of persistent structures that appear as disturbances in the background. Some of these structures are stable, so they appear as vertical lines. Others translate in space, appearing as diagonals with different slopes, depending on how many time steps they take to shift by one column. These structures are called **spaceships**.

Collisions between spaceships yield different results depending on the types of the spaceships and the phase they are in when they collide. Some collisions annihilate both ships; others leave one ship unchanged; still others yield one or more ships of different types.

These collisions are the basis of computation in a Rule 110 CA. If you think of spaceships as signals that propagate through space, and collisions as gates

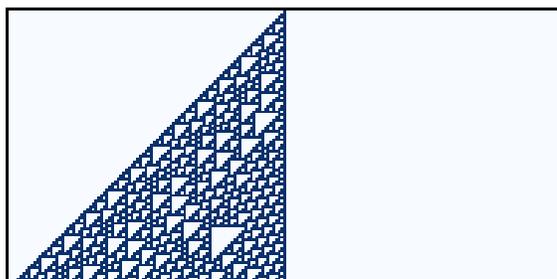


图 5.4:100 个时间步骤之后的规则 110。

这个性质，也被称为普遍性，在 1998 年被 Matthew Cook 证明，参见可计算函数 <http://thinkcomplex.com/r110>。

图 5.4 显示了规则 110 在一个单元格的初始条件和 100 个时间步骤下的样子。在这个时间尺度上，没有什么特别的事情正在发生。有一些规则的模式，但也有一些特征是难以刻画。

图 5.5 展示了一幅更大的图片，从一个随机的初始条件和 600 个时间步骤开始：

经过大约 100 个步骤后，背景沉淀成一个简单的重复模式，但是有一些持续的结构出现在背景中的干扰。其中一些结构是稳定的，所以它们看起来是垂直的线条。另一些则在空间中转换，以斜线的形式出现，斜度不同，这取决于他们移动一列所需的时间步长。这些结构被称为宇宙飞船。

太空船之间的碰撞会产生不同的结果，这取决于太空船的类型和碰撞时所处的阶段。有些碰撞会同时消灭两艘船；有些会使一艘船保持不变；还有一些会产生一艘或多艘不同类型的船。

这些碰撞是规则 110 CA 中计算的基础。如果你认为宇宙飞船是在太空中传播的信号，而碰撞就是星门

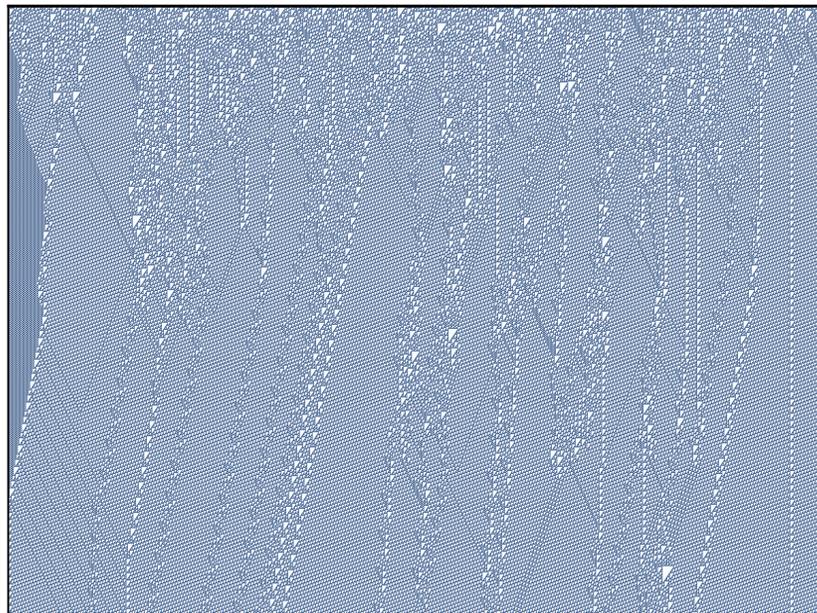


Figure 5.5: Rule 110 with random initial conditions and 600 time steps.

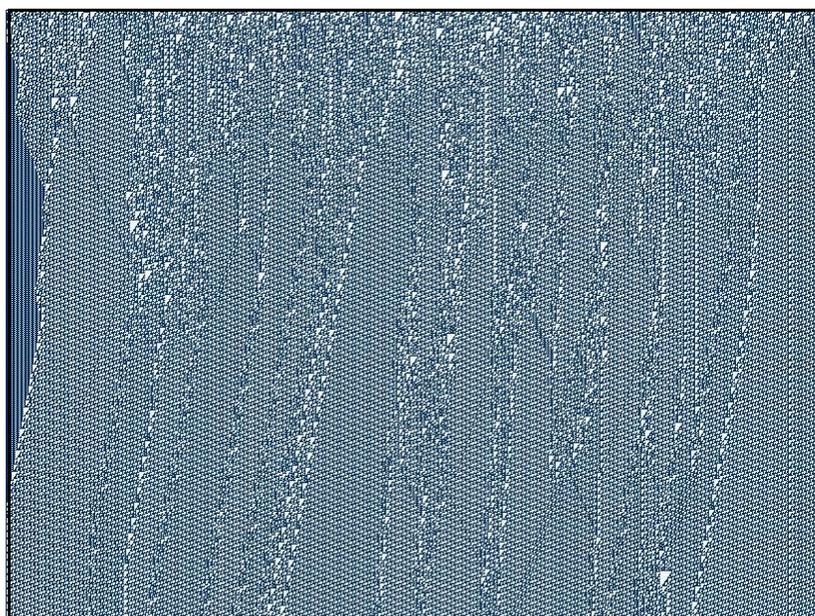


图 5.5: 带有随机初始条件和 600 个时间步骤的规则 110。

that compute logical operations like AND and OR, you can see what it means for a CA to perform a computation.

## 5.7 Universality

To understand universality, we have to understand computability theory, which is about models of computation and what they compute.

One of the most general models of computation is the Turing machine, which is an abstract computer proposed by Alan Turing in 1936. A Turing machine is a 1-D CA, infinite in both directions, augmented with a read-write head. At any time, the head is positioned over a single cell. It can read the state of that cell (usually there are only two states) and it can write a new value into the cell.

In addition, the machine has a register, which records the state of the machine (one of a finite number of states), and a table of rules. For each machine state and cell state, the table specifies an action. Actions include modifying the cell the head is over and moving one cell to the left or right.

A Turing machine is not a practical design for a computer, but it models common computer architectures. For a given program running on a real computer, it is possible (at least in principle) to construct a Turing machine that performs an equivalent computation.

The Turing machine is useful because it is possible to characterize the set of functions that can be computed by a Turing machine, which is what Turing did. Functions in this set are called “Turing computable”.

To say that a Turing machine can compute any Turing-computable function is a tautology: it is true by definition. But Turing-computability is more interesting than that.

It turns out that just about every reasonable model of computation anyone has come up with is “Turing complete”; that is, it can compute exactly the same set of functions as the Turing machine. Some of these models, like lambda calculus, are very different from a Turing machine, so their equivalence is surprising.

计算逻辑运算如 AND 和 OR，你可以看到 CA 执行计算意味着什么。

### 5.7 普遍性

为了理解普适性，我们必须理解可计算性理论，这是关于计算模型和它们计算的东西。

最普遍的计算模型之一是图灵机，它是阿兰·图灵在 1936 年提出的一种抽象计算机。图灵机是一个一维的 CA，在两个方向上都有，加上一个读写头。在任何时候，磁头都位于单个细胞之上。它可以读取该单元格的状态(通常只有两种状态)，并将一个新值写入该单元格。

此外，机器还有一个寄存器，用于记录机器的状态(状态数之一)和一个规则表。对于每个计算机状态和单元格状态，表格指定一个操作。操作包括修改磁头所在的单元格，并将一个单元格移动到左边或右边。

图灵机不是一个实际的计算机设计，但它模拟了常见的计算机体系结构。对于在实际计算机上运行的给定程序，可以(至少在原则上)构造一台执行等效计算的图灵机。

图灵机之所以有用，是因为它可以描述图灵机可以计算的一组函数，就像图灵所做的那样。这个集合中的函数称为图灵可计算”。

说图灵机可以计算任何图灵可计算函数是一种重言式：通过定义它是正确的。但是图灵可计算性更有趣。

事实证明，几乎每一个人们提出的合理的计算模型都是图灵完全”；也就是说，它可以计算出与图灵机完全相同的一组函数。其中一些模型，比如 lambda 演算，与图灵机非常不同，所以它们的等价性令人惊讶。

This observation led to the Church-Turing Thesis, which is the claim that these definitions of computability capture something essential that is independent of any particular model of computation.

The Rule 110 CA is yet another model of computation, and remarkable for its simplicity. That it, too, turns out to be Turing complete lends support to the Church-Turing Thesis.

In *A New Kind of Science*, Wolfram states a variation of this thesis, which he calls the “principle of computational equivalence” (see <http://thinkcomplex.com/equiv>):

Almost all processes that are not obviously simple can be viewed as computations of equivalent sophistication.

More specifically, the principle of computational equivalence says that systems found in the natural world can perform computations up to a maximal (“universal”) level of computational power, and that most systems do in fact attain this maximal level of computational power. Consequently, most systems are computationally equivalent.

Applying these definitions to CAs, Classes 1 and 2 are “obviously simple”. It may be less obvious that Class 3 is simple, but in a way perfect randomness is as simple as perfect order; complexity happens in between. So Wolfram’s claim is that Class 4 behavior is common in the natural world, and that almost all systems that manifest it are computationally equivalent.

## 5.8 Falsifiability

Wolfram holds that his principle is a stronger claim than the Church-Turing thesis because it is about the natural world rather than abstract models of computation. But saying that natural processes “can be viewed as computations” strikes me as a statement about theory choice more than a hypothesis about the natural world.

Also, with qualifications like “almost” and undefined terms like “obviously simple”, his hypothesis may be **unfalsifiable**. Falsifiability is an idea from

这一观察结果导致了邱奇-图灵论题计算理论，该理论认为这些可计算性的解释捕捉到了一些独立于任何特定计算模型的本质。

110ca 规则是另一种计算模型，其简单性令人瞩目。这也证明了图灵是完整的，这也为邱奇-图灵论题提供了支持。

在《一种新的科学》一书中，Wolfram 阐述了这个论点的一个变种，他称之为“计算等价原理”(见《<http://thinkcomplex.com/equiv>)：

几乎所有不明显简单的过程都可以被看作是等效复杂度的计算。

更具体地说，计算等价性原理认为，在自然世界中发现的系统可以执行最大(普遍)计算能力水平的计算，而且大多数系统实际上都达到了这一计算能力的最大水平。因此，大多数系统在计算上是等价的。

将这些规则应用于核证机关，类别 1 和类别 2 显然是简单的”。第三类可能不那么明显，但是在某种程度上，完美的随机性就像完美的顺序一样简单；复杂性发生在其中。所以 Wolfram 声称第四类行为在自然界中很常见，而且几乎所有表现出这种行为的系统在计算上都是等价的。

### 5.8 法尔西能力

认为他的原理比邱奇-图灵论题更有说服力，因为它是关于自然世界的，而不是抽象的计算模型。但是说自然过程可以被看作是计算”，我觉得这更像是关于理论选择的陈述，而不是关于自然世界的假设。

而且，由于几乎是“和显然是简单的”这样的定义，他的假设可能是不可能的。法尔西能力是一个想法来自

the philosophy of science, proposed by Karl Popper as a demarcation between scientific hypotheses and pseudoscience. A hypothesis is falsifiable if there is an experiment, at least in the realm of practicality, that would contradict the hypothesis if it were false.

For example, the claim that all life on earth is descended from a common ancestor is falsifiable because it makes specific predictions about similarities in the genetics of modern species (among other things). If we discovered a new species whose DNA was almost entirely different from ours, that would contradict (or at least bring into question) the theory of universal common descent.

On the other hand, “special creation”, the claim that all species were created in their current form by a supernatural agent, is unfalsifiable because there is nothing that we could observe about the natural world that would contradict it. Any outcome of any experiment could be attributed to the will of the creator.

Unfalsifiable hypotheses can be appealing because they are impossible to refute. If your goal is never to be proved wrong, you should choose hypotheses that are as unfalsifiable as possible.

But if your goal is to make reliable predictions about the world — and this is at least one of the goals of science — unfalsifiable hypotheses are useless. The problem is that they have no consequences (if they had consequences, they would be falsifiable).

For example, if the theory of special creation were true, what good would it do me to know it? It wouldn’t tell me anything about the creator except that he has an “inordinate fondness for beetles” (attributed to J. B. S. Haldane). And unlike the theory of common descent, which informs many areas of science and bioengineering, it would be of no use for understanding the world or acting in it.

## 5.9 What is this a model of?

Some cellular automata are primarily mathematical artifacts. They are interesting because they are surprising, or useful, or pretty, or because they provide tools for creating new mathematics (like the Church-Turing thesis).

科学哲学，由卡尔·波普尔提出，作为科学假说和伪科学之间的界限。如果有一个实验，至少在实践领域，假设如果是错误的，那么假设就是错误的。

例如，声称地球上所有生命都是从一个共同的祖先进化而来的说法是可靠的，因为它对现代物种的遗传学相似性做出了特别的预测。如果我们发现一个新物种的DNA几乎完全不同于我们的，那将与普遍共同血统理论相矛盾(或者至少引起质疑)。

另一方面，“特殊创造”，即所有物种都是由超自然力量创造出来的，这种说法是不可信的，因为我们在自然世界中观察不到任何与之相矛盾的东西。任何实验的任何结果都可以归因于创造者的意愿。

不可伪造的假设之所以吸引人，是因为它们不可能反驳。如果你的目标是永远不被证明是错误的，那么你应该选择尽可能不可能的假设。

但是如果你的目标是对这个世界做出可靠的预测，而这至少是科学的目标之一，那么假设是无用的。问题在于，它们没有任何后果(如果它们有后果，它们就是可能出错的)。

例如，如果特殊创造理论是正确的，那么知道它对我有什么好处呢？它不会告诉我任何关于创造者的事情，除了他有一个过度的甲虫爱好”(归因于约翰·伯顿·桑德森·霍尔丹)。共同血统理论指导着科学和生物工程的许多领域，而共同血统理论则不同，它对于理解这个世界或在其中采取行动毫无用处。

### 5.9 这是什么样的模型？

一些细胞自动机主要是数学工件。它们之所以有趣是因为它们令人惊讶，或者有用，或者漂亮，或者是因为它们提供了创造新数学的工具(比如邱奇-图灵论题)。

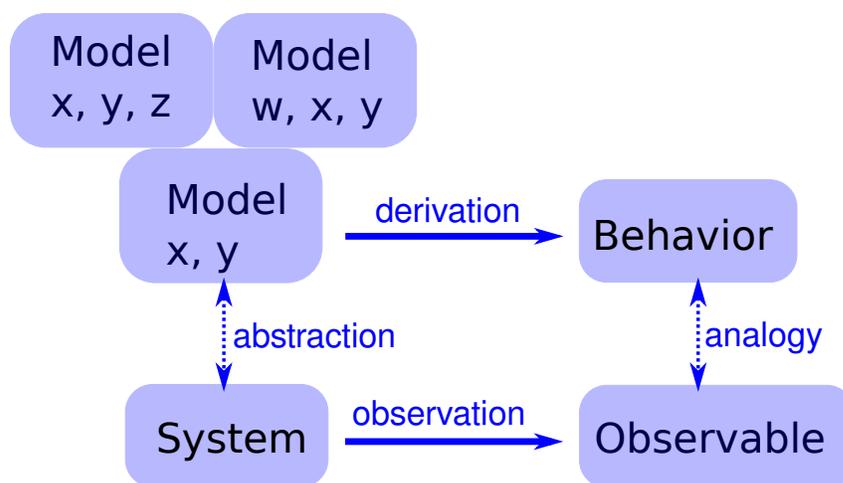


Figure 5.6: The logical structure of a simple physical model.

But it is not clear that they are models of physical systems. And if they are, they are highly abstracted, which is to say that they are not very detailed or realistic.

For example, some species of cone snail produce a pattern on their shells that resembles the patterns generated by cellular automata (see <http://thinkcomplex.com/cone>). So it is natural to suppose that a CA is a model of the mechanism that produces patterns on shells as they grow. But, at least initially, it is not clear how the elements of the model (so-called cells, communication between neighbors, rules) correspond to the elements of a growing snail (real cells, chemical signals, protein interaction networks).

For conventional physical models, being realistic is a virtue. If the elements of a model correspond to the elements of a physical system, there is an obvious analogy between the model and the system. In general, we expect a model that is more realistic to make better predictions and to provide more believable explanations.

Of course, this is only true up to a point. Models that are more detailed are harder to work with, and usually less amenable to analysis. At some point, a model becomes so complex that it is easier to experiment with the system.

At the other extreme, simple models can be compelling exactly because they are simple.

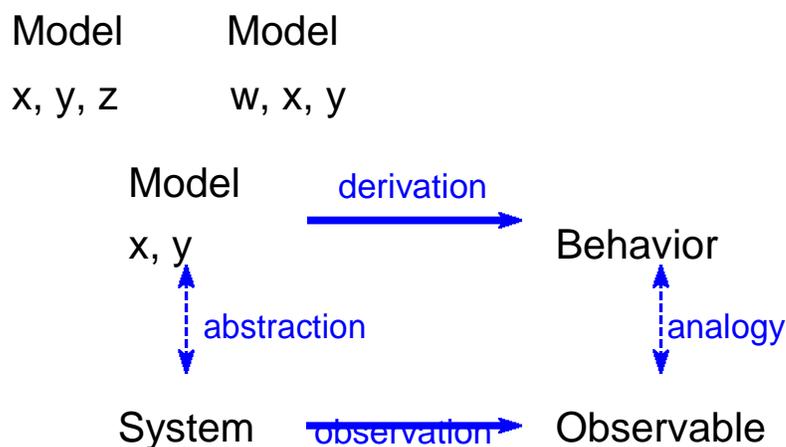


图 5.6: 简单物理模型的逻辑结构。

但它们是否是物理系统的模型尚不清楚。如果是的话，它们是高度抽象的，也就是说，它们不是非常详细或现实的。

例如，一些种类的锥形蜗牛在它们的外壳上产生一种类似于细胞自动机产生的模式的图案(见 [http:// thinkcomplex.com/cone](http://thinkcomplex.com/cone) )。因此，很自然地假设 CA 是一种在贝壳生长时在其上生成模式的机制模型。但是，至少在初始阶段，我们还不清楚这个模型的元素(所谓的细胞，邻居之间的交流，规则)是如何对应成长中的蜗牛的元素的(真正的细胞，化学信号，蛋白质相互作用网络)。

对于传统的物理模型来说，现实是一种美德。如果一个模型的元素对应于一个物理系统的元素，那么在模型和系统之间有一个明显的类比。一般来说，我们期望一个更现实的模型来做出更好的预测，并提供更可信的解释。

当然，这只是在一定程度上是正确的。更详细的模型更难使用，通常也更难进行分析。在某种程度上，模型变得如此复杂，以至于更容易在系统中进行实验。

在另一个极端，简单的模型之所以引人注目，恰恰是因为它们简单。

Simple models offer a different kind of explanation than detailed models. With a detailed model, the argument goes something like this: “We are interested in physical system  $S$ , so we construct a detailed model,  $M$ , and show by analysis and simulation that  $M$  exhibits a behavior,  $B$ , that is similar (qualitatively or quantitatively) to an observation of the real system,  $O$ . So why does  $O$  happen? Because  $S$  is similar to  $M$ , and  $B$  is similar to  $O$ , and we can prove that  $M$  leads to  $B$ .”

With simple models we can’t claim that  $S$  is similar to  $M$ , because it isn’t. Instead, the argument goes like this: “There is a set of models that share a common set of features. Any model that has these features exhibits behavior  $B$ . If we make an observation,  $O$ , that resembles  $B$ , one way to explain it is to show that the system,  $S$ , has the set of features sufficient to produce  $B$ .”

For this kind of argument, adding more features doesn’t help. Making the model more realistic doesn’t make the model more reliable; it only obscures the difference between the essential features that cause  $B$  and the incidental features that are particular to  $S$ .

Figure 5.6 shows the logical structure of this kind of model. The features  $x$  and  $y$  are sufficient to produce the behavior. Adding more detail, like features  $w$  and  $z$ , might make the model more realistic, but that realism adds no explanatory power.

## 5.10 Implementing CAs

To generate the figures in this chapter, I wrote a Python class called `Cell1D` that represents a 1-D cellular automaton, and a class called `Cell1DViewer` that plots the results. Both are defined in `Cell1D.py` in the repository for this book.

To store the state of the CA, I use a NumPy array with one column for each cell and one row for each time step.

To explain how my implementation works, I’ll start with a CA that computes the parity of the cells in each neighborhood. The “parity” of a number is 0 if the number is even and 1 if it is odd.

简单的模型比详细的模型有不同的解释。对于一个详细的模型，争论是这样的：我们对物理系统  $s$  感兴趣，所以我们构造了一个详细的模型  $m$ ，并通过分析和模拟表明  $m$  表现出一种行为， $b$ ，这种行为与对真实系统的观察相似(定性或定量)。那么  $o$  为什么会发生呢？因为  $s$  类似于  $m$ ， $b$  类似于  $o$ ，我们可以证明  $m$  导致  $b$

对于简单的模型，我们不能说  $s$  和  $m$  相似，因为它们不相似。相反，争论是这样的：有一组模型共享一组共同的特性。任何具有这些特征的模型都会表现出行为  $b$ 。如果我们做一个观察， $o$ ，类似于  $b$ ，解释它的一种方法是证明系统  $s$  具有产生  $b$  的一系列特征

对于这种论点，增加更多的功能是没有帮助的。使模型更加真实并不能使模型更加可靠；它只是掩盖了导致  $b$  的基本特征和  $s$  特有的附带特征之间的差异。

图 5.6 显示了这种模型的逻辑结构。特性  $x$  和  $y$  是产生行为的绝对依据。增加更多的细节，比如特征  $w$  和  $z$ ，可能会使模型更加真实，但是这种真实感并不会增加任何解释力。

### 5.10 推行核证机关

为了在本章中生成 `gures`，我编写了一个名为 `Cell1D` 的 Python 类，它表示 1-D 细胞自动机，还有一个名为 `Cell1DViewer` 的类，它绘制结果。两者都存放在本书的仓库 `Cell1D.py` 中。

为了存储 CA 的状态，我使用 NumPy 数组，每个单元格有一列，每个时间步骤有一行。

为了解释我的实现是如何工作的，我将从一个 CA 开始，该 CA 计算每个邻居中的单元格的奇偶校验。如果一个数是偶数，其奇偶性为 0；如果是奇数，则为 1。

I use the NumPy function `zeros` to create an array of zeros, then put a 1 in the middle of the first row.

```
rows = 5
cols = 11
array = np.zeros((rows, cols), dtype=np.uint8)
array[0, 5] = 1
print(array)

[[ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

The data type `uint8` indicates that the elements of `array` are unsigned 8-bit integers.

`plot_ca` displays the elements of an `array` graphically:

```
import matplotlib.pyplot as plt

def plot_ca(array, rows, cols):
    cmap = plt.get_cmap('Blues')
    plt.imshow(array, cmap=cmap, interpolation='none')
```

I import `pyplot` with the abbreviated name `plt`, which is conventional. The function `get_cmap` returns a colormap, which maps from the values in the array to colors. The colormap `'Blues'` draws the “on” cells in dark blue and the “off” cells in light blue.

`imshow` displays the array as an “image”; that is, it draws a colored square for each element of the array. Setting `interpolation` to `none` indicates that `imshow` should not interpolate between on and off cells.

To compute the state of the CA during time step `i`, we have to add up consecutive elements of `array` and compute the parity of the sum. We can do that using a slice operator to select the elements and the modulus operator to compute parity:

我使用 NumPy 函数的零来创建一个零数组，然后将 1 放在第一行的中间。

```
行 = 5
11
Array = np.zeros ((rows, cols) , dtype = np.uint8)
Array [0,5] = 1
打印(数组)

[0.0.0.0.0.1.0.0.0.0.0.0.]
[ 0.0.0.0.0.0.0.0.0.0. ]
[ 0.0.0.0.0.0.0.0.0.0. ]
[ 0.0.0.0.0.0.0.0.0.0. ]
[ 0.0.0.0.0.0.0.0.0.[0]
```

数据类型 `uint8` 表示数组的元素是无符号的 8 位整数。

以图形方式显示数组的元素:

```
导入 matplotlib.pyplot 作为 plt

Def plot _ ca (array, rows, cols) :
    Cmap = plt.get _ cmap (' Blues')
    Imshow (数组, cmap = cmap, 插值 = ' none')
```

我导入了带有缩写名称 `plt` 的 `pyplot`，这是常规的。函数 `get_cmap` 返回一个 `colormap`，它从数组中的值映射到颜色。颜色图“蓝色”用深蓝色绘制“单元格”，用浅蓝色绘制“o”单元格。

`Imshow` 将数组显示为图像”；也就是说，它为数组的每个元素绘制一个有色的正方形。将插值设置为 `none` 表示 `imshow` 不应插值到和 o 单元格之间。

为了计算 CA 在时间步骤 `i` 中的状态，我们必须加上数组的连续元素并计算和的奇偶校验。我们可以用一个切片运算符来选择元素，用模运算符来计算奇偶校验:

```
def step(array, i):
    rows, cols = array.shape
    row = array[i-1]
    for j in range(1, cols):
        elts = row[j-1:j+2]
        array[i, j] = sum(elts) % 2
```

`rows` and `cols` are the dimensions of the array. `row` is the previous row of the array.

Each time through the loop, we select three elements from `row`, add them up, compute the parity, and store the result in row `i`.

In this example, the lattice is finite, so the first and last cells have only one neighbor. To handle this special case, I don't update the first and last column; they are always 0.

## 5.11 Cross-correlation

The operation in the previous section — selecting elements from an array and adding them up — is an example of an operation that is so useful, in so many domains, that it has a name: **cross-correlation**. And NumPy provides a function, called `correlate`, that computes it. In this section I'll show how we can use NumPy to write a simpler, faster version of `step`.

The NumPy `correlate` function takes an array,  $a$ , and a “window”,  $w$ , with length  $N$  and computes a new array,  $c$ , where element  $k$  is the following summation:

$$c_k = \sum_{n=0}^{N-1} a_{n+k} \cdot w_n$$

We can write this operation in Python like this:

```
def c_k(a, w, k):
    N = len(w)
    return sum(a[k:k+N] * w)
```

This function computes element  $k$  of the correlation between  $a$  and  $w$ . To show how it works, I'll create an array of integers:

```

Def step (array, i):
    一排排, 一排排, 一排排
    Row = array [ i-1]
    对于 j in range (1, cols):
        Elts = row [ j-1: j + 2]
        Array [ i, j ] = sum (elts)% 2

```

Row 和 cols 是数组的维数, row 是数组的前一行。

每次通过循环, 我们从行中选择三个元素, 将它们加起来, 计算奇偶校验, 并将结果存储在行 i 中。

在这个例子中, 格子是 nite, 所以第一个和最后一个单元格只有一个邻居。为了处理这种特殊情况, 我不更新第一列和最后一列; 它们总是 0。

### 5.11 互相关

上一节 | 从数组中选择元素并将它们加起来 | 的操作就是这样一个例子, 这个操作在很多领域都非常有用, 以至于它有一个名字: 互相关。NumPy 提供了一个名为 correlate 的函数来计算它。在本节中, 我将展示如何使用 NumPy 来编写更简单、更快速的步骤。

函数接受一个长度为 n 的数组 a 和一个窗口 w, 并计算一个新的数组 c, 其中元素 k 是以下的总和:

$$C_k = \sum_{n=0}^{N-1-k} a_{n+k} w_n$$

我们可以这样用 Python 编写这个操作:

```

(a, w, k):
    N = len (w)
    返回和(a [ k: k + n ] * w)

```

这个函数计算元素 a 和 w 之间的相关性, 为了说明它是如何工作的, 我将创建一个整数数组:

```
N = 10
row = np.arange(N, dtype=np.uint8)
print(row)

[0 1 2 3 4 5 6 7 8 9]
```

And a window:

```
window = [1, 1, 1]

print(window)
```

With this window, each element, `c_k`, is the sum of consecutive elements from `a`:

```
c_k(row, window, 0)
3

c_k(row, window, 1)
6
```

We can use `c_k` to write `correlate`, which computes the elements of `c` for all values of `k` where the window and the array overlap.

```
def correlate(row, window):
    cols = len(row)
    N = len(window)
    c = [c_k(row, window, k) for k in range(cols-N+1)]
    return np.array(c)
```

Here's the result:

```
c = correlate(row, window)
print(c)

[ 3  6  9 12 15 18 21 24]
```

The NumPy function `correlate` does the same thing:

```
c = np.correlate(row, window, mode='valid')
print(c)

[ 3  6  9 12 15 18 21 24]
```

```

10
Row = np.arange (n, dtype = np.uint8)
打印(行)

[0123456789]

```

还有一扇窗:

```

Window = [1,1,1]

打印(窗口)

```

在这个窗口中，每个元素  $c_k$  都是  $a$  中连续元素的和:

```

行, 窗, 0)
图 3

行, 窗, 1)
图 6

```

我们可以使用  $c_k$  来编写 `correlate`，它为窗口和数组重叠的  $k$  的所有值计算  $c$  的元素。

```

Def correlate (行、窗口):
    Cols = len (row)
    (窗口)
    C = [ c_k (row, window, k) for k in range (cols-N + 1)]返回
    np.array (c)

```

结果如下:

```

相关联(行, 窗口)
列印(c)

[3691215182124]

```

函数关联起来做同样的事情:

```

C = np.correlate (row, window, mode = ' valid') print (c)

```

The argument `mode='valid'` means that the result contains only the elements where the window and array overlap, which are considered valid.

The drawback of this mode is that the result is not the same size as `array`. We can fix that with `mode='same'`, which adds zeros to the beginning and end of `array`:

```
c = np.correlate(row, window, mode='same')
print(c)

[ 1  3  6  9 12 15 18 21 24 17]
```

Now the result is the same size as `array`. As an exercise at the end of this chapter, you'll have a chance to write a version of `correlate` that does the same thing.

We can use NumPy's implementation of `correlate` to write a simple, faster version of `step`:

```
def step2(array, i, window=[1,1,1]):
    row = array[i-1]
    c = np.correlate(row, window, mode='same')
    array[i] = c % 2
```

In the notebook for this chapter, you'll see that `step2` yields the same results as `step`.

## 5.12 CA tables

The function we have so far works if the CA is “totalitic”, which means that the rules only depend on the sum of the neighbors. But most rules also depend on which neighbors are on and off. For example, 100 and 001 have the same sum, but for many CAs, they would yield different results.

We can make `step2` more general using a window with elements `[4, 2, 1]`, which interprets the neighborhood as a binary number. For example, the neighborhood 100 yields 4; 010 yields 2, and 001 yields 1. Then we can take these results and look them up in the rule table.

参数 `mode = "valid"` 意味着结果只包含窗口和数组重叠的元素，这些元素被认为是有效的。

这种模式的缺点是结果与数组的大小不一样。我们可以用 `mode = 'same'` 来 `x`，这样可以在数组的开头和结尾添加零：

```
C = np.correlate(row, window, mode = 'same') print
(c)

[1369121518212417]
```

现在，结果是与数组一样的大小。作为本章最后的练习，您将有机会编写一个与之相关的版本。

我们可以使用 NumPy 的 `correlate` 实现来编写一个简单、快速的步骤：

```
2(array, i, window = [1,1,1]):
    Row = array [ i-1]
    C = np.correlate(row, window, mode = 'same') array
    [ i] = c% 2
```

在本章的笔记本中，您将看到步骤 2 产生了与步骤相同的结果。

### 5.12 个核证机关表

如果元胞自动机是完全连续的，那么我们现有的函数是有效的”，这意味着规则只依赖于邻居的和。但是，大多数规则也取决于哪些邻居是 `on` 和 `o`。例如，`100` 和 `001` 具有相同的总和，但对于许多 `ca`，它们将产生不同的结果。

我们可以使用一个包含元素 `[4,2,1]` 的窗口使步骤 2 更加通用，它将这个邻域解释为一个二进制数。例如，邻近的 `100` 产生 4；`010` 产生 2，`001` 产生 1。然后我们可以获得这些结果，并在规则表中查找它们。

Here's the more general version of `step2`:

```
def step3(array, i, window=[4,2,1]):
    row = array[i-1]
    c = np.correlate(row, window, mode='same')
    array[i] = table[c]
```

The first two lines are the same. Then the last line looks up each element from `c` in `table` and assigns the result to `array[i]`.

Here's the function that computes the table:

```
def make_table(rule):
    rule = np.array([rule], dtype=np.uint8)
    table = np.unpackbits(rule)[::-1]
    return table
```

The parameter, `rule`, is an integer between 0 and 255. The first line puts `rule` into an array with a single element so we can use `unpackbits`, which converts the rule number to its binary representation. For example, here's the table for Rule 150:

```
>>> table = make_table(150)
>>> print(table)
[0 1 1 0 1 0 0 1]
```

The code in this section is encapsulated in the `Cell11D` class, defined in `Cell11D.py` in the repository for this book.

## 5.13 Exercises

The code for this chapter is in the Jupyter notebook `chap05.ipynb` in the repository for this book. Open this notebook, read the code, and run the cells. You can use this notebook to work on the exercises in this chapter. My solutions are in `chap05soln.ipynb`.

**Exercise 5.1** Write a version of `correlate` that returns the same result as `np.correlate` with `mode='same'`. Hint: use the NumPy function `pad`.

下面是步骤 2 的一般版本:

```
3(array, i, window = [4,2,1]):  
    Row = array [ i-1]  
    C = np.correlate (row, window, mode = ' same') array  
    [ i ] = table [ c ]
```

第一行和第二行是一样的。然后最后一行从表中的 c 查找每个元素，并将结果分配给 array [ i ]。

下面是计算表的函数:

```
Def make _ table (rule) :  
    Rule = np.array ([ rule ] , dtype = np.uint8) table =  
    np.unpackbits (rule)[ :-1] return table
```

参数 rule 是介于 0 和 255 之间的整数。第一行将规则放入一个只有一个元素的数组中，这样我们就可以使用 `unpackbits`，它将规则数转换为它的二进制表示形式。例如，下面是规则 150 的表格:

```
>>> Table = make _ table (150)  
>>> 列印(表格)  
[01101001]
```

本节中的代码封装在 `Cell1D` 类中，在本书的仓库中的 `Cell1D.py` 中进行了解释。

### 5.13 练习

本章的代码在本书资料库中的 `chapyter` 笔记本 `chap05.ipynb` 中。打开这个笔记本，阅读代码，并运行单元格。你可以用这个笔记本来做这一章的练习。我的解决方案在第 5 章 `soln`。

练习 5.1 编写一个 `correlate` 版本，它返回与 `np` 相同的结果，并且与 `mode = '相同'` 相关。提示: 使用 `NumPy` 函数垫。

**Exercise 5.2** This exercise asks you to experiment with Rule 110 and some of its spaceships.

1. Read the Wikipedia page about Rule 110, which describes its background pattern and spaceships: <http://thinkcomplex.com/r110>.
2. Create a Rule 110 CA with an initial condition that yields the stable background pattern.  
Note that the `Cell1D` class provides `start_string`, which allows you to initialize the state of the array using a string of 1s and 0s.
3. Modify the initial condition by adding different patterns in the center of the row and see which ones yield spaceships. You might want to enumerate all possible patterns of  $n$  bits, for some reasonable value of  $n$ . For each spaceship, can you find the period and rate of translation? What is the biggest spaceship you can find?
4. What happens when spaceships collide?

**Exercise 5.3** The goal of this exercise is to implement a Turing machine.

1. Read about Turing machines at <http://thinkcomplex.com/tm>.
2. Write a class called `Turing` that implements a Turing machine. For the action table, use the rules for a 3-state busy beaver.
3. Write a class named `TuringViewer` that generates an image that represents the state of the tape and the position and state of the head. For one example of what that might look like, see <http://thinkcomplex.com/turing>.

**Exercise 5.4** This exercise asks you to implement and test several PRNGs. For testing, you will need to install `DieHarder`, which you can download from <http://thinkcomplex.com/dh>, or it might be available as a package for your operating system.

1. Write a program that implements one of the linear congruential generators described at <http://thinkcomplex.com/lcg>. Test it using `DieHarder`.
2. Read the documentation of Python's `random` module. What PRNG does it use? Test it.

练习 5.2 这个练习要求你试验规则 110 和它的一些宇宙飞船。

1. 阅读关于 110 规则的维基百科页面，其中描述了它的背景图案和宇宙飞船：  
<http://thinkcomplex.com/r110>。
2. 创建一个规则 110 CA 与一个初始条件，产生稳定的背景模式。  
注意，Cell1D 类提供 start 字符串，它允许您使用 1 和 0 的字符串初始化数组的状态。
3. 修改初始条件，在行的中心添加不同的图案，看看哪些图案会产生宇宙飞船。你可能想列举所有可能的  $n$  位模式，对于一些合理的  $n$  值，你能计算出每艘飞船的转换周期和速率吗？你能找到的最大的宇宙飞船是什么？
4. 当宇宙飞船相撞时会发生什么？

练习 5.3 这个练习的目标是实现一个图灵机。

1. 阅读关于图灵机的 <http://thinkcomplex.com/tm>。
2. 编写一个名为 Turing 的类来实现一个 Turing 机。对于行动表，使用规则为三个州繁忙的海狸。
3. 编写一个名为 TuringViewer 的类，该类生成一个表示磁带状态以及磁头位置和状态的图像。举个例子来说明这看起来是什么样子的，参见《<http://thinkcomplex.com/turing>》。

练习 5.4 这个练习要求你实施和测试几个 PRNGs。对于测试，你需要安装 DieHarder，你可以从 <http://thinkcomplex.com/dh> 下载，或者它可以作为你的操作系统的一个软件包。

1. 编写一个程序来实现 <http://thinkcomplex.com/lcg> 中描述的线性同余发生器，然后使用 DieHarder 进行测试。
2. 阅读 Python 的 random 模块的文档。它使用什么 PRNG? 测试它。

3. Implement a Rule 30 CA with a few hundred cells, run it for as many time steps as you can in a reasonable amount of time, and output the center column as a sequence of bits. Test it.

**Exercise 5.5** Falsifiability is an appealing and useful idea, but among philosophers of science it is not generally accepted as a solution to the demarcation problem, as Popper claimed.

Read <http://thinkcomplex.com/false> and answer the following questions.

1. What is the demarcation problem?
2. How, according to Popper, does falsifiability solve the demarcation problem?
3. Give an example of two theories, one considered scientific and one considered unscientific, that are successfully distinguished by the criterion of falsifiability.
4. Can you summarize one or more of the objections that philosophers and historians of science have raised to Popper's claim?
5. Do you get the sense that practicing philosophers think highly of Popper's work?

3. 用几百个单元格实现一个 Rule 30 CA，在合理的时间内尽可能多地运行它，并以位序列的形式输出中心列。测试一下。

练习 5.5 Falsi 能力是一个吸引人的有用的想法，但是在科学哲学家中，它并不像 Popper 声称的那样被普遍接受为画界问题的解决方案。

阅读 <http://thinkcomplex.com/false> 并回答以下问题。

1. 什么是画界问题？
2. 根据波普尔的观点，伪装能力如何解决画界问题问题？
3. 举例说明两种理论，一种被认为是科学理论，另一种被认为是非科学理论，这两种理论成功地以假说能力为标准加以区分。
4. 你能概括一个或多个科学哲学家和历史学家对波普尔的主张提出的反对意见吗？
5. 你有没有感觉到实践哲学家对波普的作品评价很高？





# Chapter 6

## Game of Life

In this chapter we consider two-dimensional cellular automata, especially John Conway’s Game of Life (GoL). Like some of the 1-D CAs in the previous chapter, GoL follows simple rules and produces surprisingly complicated behavior. And like Wolfram’s Rule 110, GoL turns out to be universal; that is, it can compute any computable function, at least in theory.

Complex behavior in GoL raises issues in the philosophy of science, particularly related to scientific realism and instrumentalism. I discuss these issues and suggest additional reading.

At the end of the chapter, I demonstrate ways to implement GoL efficiently in Python.

The code for this chapter is in `chap06.ipynb` in the repository for this book. More information about working with the code is in Section 0.3.

### 6.1 Conway’s GoL

One of the first cellular automata to be studied, and probably the most popular of all time, is a 2-D CA called “The Game of Life”, or GoL for short. It was developed by John H. Conway and popularized in 1970 in Martin Gardner’s column in *Scientific American*. See <http://thinkcomplex.com/gol>.

## 第六章

### 生命的游戏

在这一章中，我们考虑二维元胞自动机，特别是约翰康威的生命游戏(GoL)。与前一章中的一些一维 `ca` 一样，GoL 遵循简单的规则，并产生出奇复杂的行为。就像 Wolfram 的 110 定律一样，GoL 被证明是通用的；也就是说，它可以计算任何可计算函数，至少在理论上是这样。

语言中的复杂行为引发了科学哲学的问题，特别是与科学实在论和工具主义有关的问题。我讨论了这些问题，并建议进一步阅读。

在本章的最后，介绍了在 Python 中实现 GoL 的方法。

本章的代码位于本书知识库中的 `chap06.ipynb` 中。

关于使用代码的更多信息请参见 0.3 部分。

#### 6.1 Conway's GoL

第一个被研究的细胞自动机，也许是有史以来最流行的一个，是一个叫做生命游戏的二维 CA，简称 GoL。它由 John h. Conway 开发，并于 1970 年在 Martin Gardner 在 *Scienti* 的专栏中推广。参见 <http://thinkcomplex.com/gol>。

The cells in GoL are arranged in a 2-D **grid**, that is, an array of rows and columns. Usually the grid is considered to be infinite, but in practice it is often “wrapped”; that is, the right edge is connected to the left, and the top edge to the bottom.

Each cell in the grid has two states — live and dead — and 8 neighbors — north, south, east, west, and the four diagonals. This set of neighbors is sometimes called a “Moore neighborhood”.

Like the 1-D CAs in the previous chapters, GoL evolves over time according to rules, which are like simple laws of physics.

In GoL, the next state of each cell depends on its current state and its number of live neighbors. If a cell is alive, it stays alive if it has 2 or 3 neighbors, and dies otherwise. If a cell is dead, it stays dead unless it has exactly 3 neighbors.

This behavior is loosely analogous to real cell growth: cells that are isolated or overcrowded die; at moderate densities they flourish.

GoL is popular because:

- There are simple initial conditions that yield surprisingly complex behavior.
- There are many interesting stable patterns: some oscillate (with various periods) and some move like the spaceships in Wolfram’s Rule 110 CA.
- And like Rule 110, GoL is Turing complete.
- Another factor that generated interest was Conway’s conjecture — that there is no initial condition that yields unbounded growth in the number of live cells — and the \$50 bounty he offered to anyone who could prove or disprove it.
- Finally, the increasing availability of computers made it possible to automate the computation and display the results graphically.

GoL 中的单元格排列在二维网格中，即由行和列组成的数组。通常认为网格是黑暗的，但在实践中，它往往是“包装”，也就是说，右边缘连接到左边，顶部边缘到底部。

网格中的每个单元格有两个状态 | 活的和死的 | 以及 8 个邻居 | 北、南、东、西和四条对角线。这一组邻居有时被称为摩尔社区。”。

与前面章节中的 1-D ca 一样，GoL 也是按照简单的物理定律随时间演化的。

在 GoL 中，每个细胞的下一个状态取决于它的当前状态和它的活邻居的数量。如果一个细胞是活的，如果它有 2 或 3 个邻居，它就会保持活性，否则就会死亡。如果一个细胞死亡了，那么它将一直死亡，除非它有 3 个相邻的细胞。

这种行为与真正的细胞生长有一定的相似性：分离或过度拥挤的细胞死亡；在中等密度下，它们繁殖。

GoL 之所以流行是因为：

有简单的初始条件，产生惊人的复杂是-  
哈维尔。

有许多有趣的稳定模式：一些振荡(有不同的周期)，一些像 Wolfram 的 110 CA 规则中的宇宙飞船一样移动。

和规则 110 一样，GoL 是图灵完整的。

另一个引起人们兴趣的因素是康威的猜想 | 没有一个初始条件能使活细胞的数量无限增长 | 以及他给任何能证明这一点的人提供的 50 美元的赏金或者推翻它。

最后，计算机的可用性不断增加，使计算自动化和以图形方式显示结果成为可能。

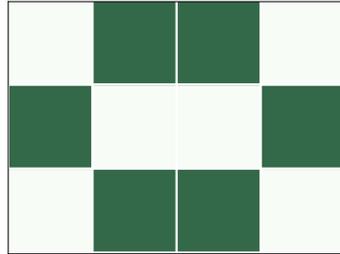


Figure 6.1: A stable pattern called a beehive.

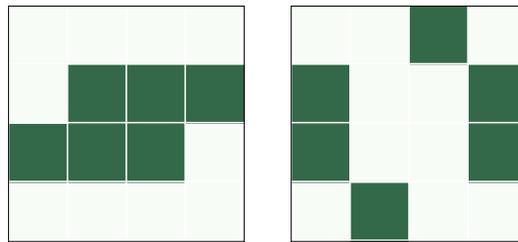


Figure 6.2: An oscillator called a toad.

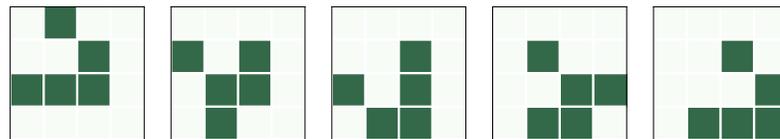


Figure 6.3: A spaceship called a glider.

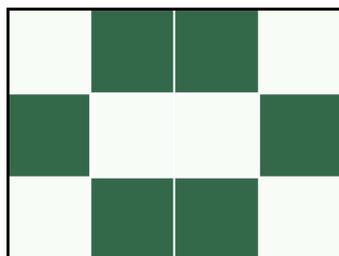


图 6.1: 称为蜂巢的稳定模式。

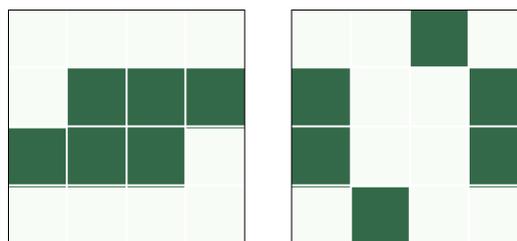


图 6.2: 一种叫蟾蜍的振荡器。

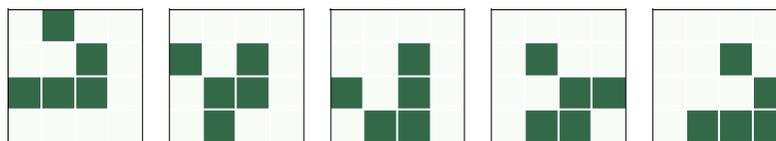


图 6.3: 一个叫做滑翔机的宇宙飞船。

## 6.2 Life patterns

If you run GoL from a random starting state, a number of stable patterns are likely to appear. Over time, people have identified these patterns and given them names.

For example, Figure 6.1 shows a stable pattern called a “beehive”. Every cell in the beehive has 2 or 3 neighbors, so they all survive, and none of the dead cells adjacent to the beehive has 3 neighbors, so no new cells are born.

Other patterns “oscillate”; that is, they change over time but eventually return to their starting configuration (provided they don’t collide with another pattern). For example, Figure 6.2 shows a pattern called a “toad”, which is an oscillator that alternates between two states. The “period” of this oscillator is 2.

Finally, some patterns oscillate and return to the starting configuration, but shifted in space. Because these patterns seem to move, they are called “spaceships”.

Figure 6.3 shows a spaceship called a “glider”. After a period of 4 steps, the glider is back in the starting configuration, shifted one unit down and to the right.

Depending on the starting orientation, gliders can move along any of the four diagonals. There are other spaceships that move horizontally and vertically.

People have spent embarrassing amounts of time finding and naming these patterns. If you search the web, you will find many collections.

## 6.3 Conway’s conjecture

From most initial conditions, GoL quickly reaches a stable state where the number of live cells is nearly constant (possibly with some oscillation).

But there are some simple starting conditions that yield a surprising number of live cells, and take a long time to settle down. Because these patterns are so long-lived, they are called “Methuselahs”.

### 6.2 生活模式

如果您从随机起始状态运行 GoL，则可能会出现许多稳定的模式。随着时间的推移，人们已经认识了这些模式，并给它们起了名字。

例如，图 6.1 显示了一个称为蜂巢的稳定模式”。蜂巢中的每个细胞都有 2 到 3 个相邻的细胞，所以它们都能存活下来，而且靠近蜂巢的死细胞中没有一个有 3 个相邻的细胞，所以没有新细胞诞生。

也就是说，它们会随着时间的推移而改变，但最终会回到它们开始的状态(前提是它们不会与其他模式发生冲突)。例如，图 6.2 显示了一种称为蟾蜍的模式，蟾蜍是一种在两种状态之间交替的振荡器。这个振荡器的周期是 2。

最后，一些模式振荡并返回到起始状态，但在空间中发生了变化。因为这些图案似乎在移动，所以它们被称为宇宙飞船”。

图 6.3 显示了一个叫做滑翔机的宇宙飞船。经过 4 个阶段后，滑翔机又回到了起始状态，向右移动了一个单位。

取决于起始方向，滑翔机可以沿着四条对角线中的任何一条移动。还有其他的宇宙飞船可以水平和垂直移动。

人们已经花费了大量的时间来理解和命名这些模式。如果你在网上搜索，你会找到许多收藏品。

### 6.3 康威猜想

从大多数初始条件来看，GoL 很快达到一个稳定的状态，其中活细胞的数量几乎是恒定的(可能有一些振荡)。

但是有一些简单的起始条件可以产生数量惊人的活细胞，并且需要很长时间才能稳定下来。因为这些模式是如此长寿，它们被称为玛士撒拉”。

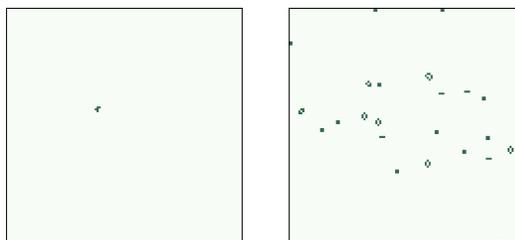


Figure 6.4: Starting and final configurations of the r-pentomino.

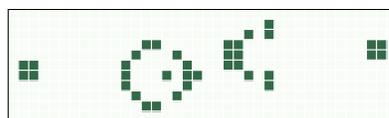


Figure 6.5: Gosper's glider gun, which produces a stream of gliders.

One of the simplest Methuselaha is the r-pentomino, which has only five cells, roughly in the shape of the letter “r”. Figure 6.4 shows the initial configuration of the r-pentomino and the final configuration after 1103 steps.

This configuration is “final” in the sense that all remaining patterns are either stable, oscillators, or gliders that will never collide with another pattern. In total, the r-pentomino yields 6 gliders, 8 blocks, 4 blinkers, 4 beehives, 1 boat, 1 ship, and 1 loaf.

The existence of long-lived patterns prompted Conway to wonder if there are initial patterns that never stabilize. He conjectured that there were not, but he described two kinds of pattern that would prove him wrong, a “gun” and a “puffer train”. A gun is a stable pattern that periodically produces a spaceship — as the stream of spaceships moves out from the source, the number of live cells grows indefinitely. A puffer train is a translating pattern that leaves live cells in its wake.

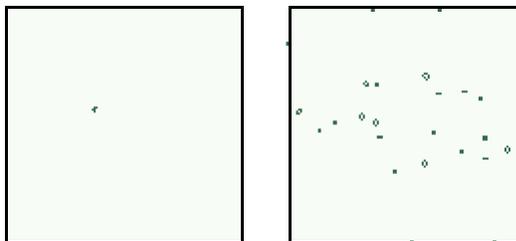


图 6.4: r-pentomino 的开始和终结。

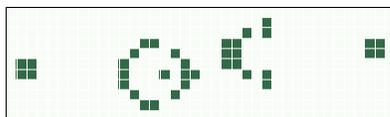


图 6.5: 高斯帕的滑翔机枪，它能产生一系列滑翔机。

最简单的玛士撒拉是 r-pentomino，它只有 5 个细胞，大致是字母 r 的形状。图 6.4 显示了 r-pentomino 的初始饱和度和 1103 步之后的饱和度。

从某种意义上说，所有剩余的模式要么是稳定的，要么是振荡器，要么是永远不会与其他模式碰撞的滑翔机。总的来说，r-pentomino 生产 6 架滑翔机，8 个滑翔机，4 个闪光灯，4 个蜂巢，1 艘船，1 艘船和 1 条面包。

长寿命模式的存在促使 Conway 怀疑是否存在永远不会稳定的初始模式。他推测没有，但是他描述了两种可以证明他错了的模式，一把“枪”和一系列“普洱火车”。枪是一种周期性产生宇宙飞船的稳定模式

| 随着宇宙飞船从源头向外移动，活细胞的数量会逐渐增加。普洱火车是一种翻译模式，在它的尾流中留下活细胞。

It turns out that both of these patterns exist. A team led by Bill Gosper discovered the first, a glider gun now called Gosper's Gun, which is shown in Figure 6.5. Gosper also discovered the first puffer train.

There are many patterns of both types, but they are not easy to design or find. That is not a coincidence. Conway chose the rules of GoL so that his conjecture would not be obviously true or false. Of all possible rules for a 2-D CA, most yield simple behavior: most initial conditions stabilize quickly or grow unboundedly. By avoiding uninteresting CAs, Conway was also avoiding Wolfram's Class 1 and Class 2 behavior, and probably Class 3 as well.

If we believe Wolfram's Principle of Computational Equivalence, we expect GoL to be in Class 4, and it is. The Game of Life was proved Turing complete in 1982 (and again, independently, in 1983). Since then, several people have constructed GoL patterns that implement a Turing machine or another machine known to be Turing complete.

## 6.4 Realism

Stable patterns in GoL are hard not to notice, especially the ones that move. It is natural to think of them as persistent entities, but remember that a CA is made of cells; there is no such thing as a toad or a loaf. Gliders and other spaceships are even less real because they are not even made up of the same cells over time. So these patterns are like constellations of stars. We perceive them because we are good at seeing patterns, or because we have active imaginations, but they are not real.

Right?

Well, not so fast. Many entities that we consider "real" are also persistent patterns of entities at a smaller scale. Hurricanes are just patterns of air flow, but we give them personal names. And people, like gliders, are not made up of the same cells over time.

This is not a new observation — about 2500 years ago Heraclitus pointed out that you can't step in the same river twice — but the entities that appear in the Game of Life are a useful test case for thinking about scientific realism.

事实证明，这两种模式都存在。由比尔·高斯帕领导的研究小组发现了第一种滑翔机枪，现在称为高斯帕的枪，如图 6.5 所示。高斯帕还发现了第一列普洱火车。

这两种类型都有许多模式，但它们不容易设计或查找。这不是巧合。康威选择了 GoL 的规则，这样他的猜想就不会明显地是对还是错。在一个二维 CA 的所有可能的规则中，大多数产生了简单的行为：大多数初始条件很快地稳定下来或者无限地增长。通过避免无趣的 ca，Conway 也避免了 Wolfram 的 1 类和 2 类行为，可能还有 3 类行为。

如果我们相信 Wolfram 的计算等价性原理，我们期望 GoL 是第四类，事实也的确如此。《生命的游戏》在 1982 年被证明是图灵完成的(1983 年又被证明是独立完成的)。从那时起，一些人已经构建了 GoL 模式，实现了一个图灵机或另一个已知的图灵完成机。

#### 6.4 现实主义

稳定的模式很难不被注意到，特别是那些移动的模式。人们很自然地认为它们是持久存在的实体，但请记住，CA 是由细胞构成的，没有蟾蜍或面包这样的东西。滑翔机和其他宇宙飞船甚至更不真实，因为随着时间的推移，它们甚至不是由相同的细胞组成的。所以这些图案就像星座。我们感知它们是因为我们善于看到模式，或者因为我们有活跃的想象力，但它们不是真实的。

对吧？

好吧，没那么快。许多我们认为是真实的实体”也是较小规模实体的持久模式。飓风只是空气的模式，但我们给它们起了个人名字。而人类，就像滑翔机一样，随着时间的推移，并不是由相同的细胞组成的。

这不是一个新的观察 | 大约 2500 年前赫拉克利特指出，你不能在同一条河里走两次 | 但是出现在《生命的游戏》中的实体是一个思考科学现实主义的有用的测试案例。

**Scientific realism** pertains to scientific theories and the entities they postulate. A theory postulates an entity if it is expressed in terms of the properties and behavior of the entity. For example, theories about electromagnetism are expressed in terms of electrical and magnetic fields. Some theories about economics are expressed in terms of supply, demand, and market forces. And theories about biology are expressed in terms of genes.

But are these entities real? That is, do they exist in the world independent of us and our theories?

Again, I find it useful to state philosophical positions in a range of strengths; here are four statements of scientific realism with increasing strength:

**SR1:** Scientific theories are true or false to the degree that they approximate reality, but no theory is exactly true. Some postulated entities may be real, but there is no principled way to say which ones.

**SR2:** As science advances, our theories become better approximations of reality. At least some postulated entities are known to be real.

**SR3:** Some theories are exactly true; others are approximately true. Entities postulated by true theories, and some entities in approximate theories, are real.

**SR4:** A theory is true if it describes reality correctly, and false otherwise. The entities postulated by true theories are real; others are not.

SR4 is so strong that it is probably untenable; by such a strict criterion, almost all current theories are known to be false. Most realists would accept something in the range between SR1 and SR3.

## 6.5 Instrumentalism

But SR1 is so weak that it verges on **instrumentalism**, which is the view that theories are instruments that we use for our purposes: a theory is useful, or not, to the degree that it is fit for its purpose, but we can't say whether it is true or false.

To see whether you are comfortable with instrumentalism, I made up the following test. Read the following statements and give yourself a point for each one you agree with. If you score 4 or more, you might be an instrumentalist!

科学实在论与科学理论及其假设的实体有关。一个理论假设一个实体，如果它是用实体的性质和行为来表示的话。例如，关于电磁学的理论是用电场和磁场来表示的。有些经济学理论是用供给、需求和市场力量来表达的。有关生物学的理论是用基因来表达的。

但是，这些实体是真实的吗？也就是说，它们是否独立于我们和我们的理论而存在于这个世界？

再一次，我认为陈述哲学立场在一系列的力量是有用的；这里有四个声明的科学实在论越来越强大：

科学理论的真或假取决于它们接近现实的程度，但没有一个理论是完全正确的。一些假设的实体可能是真实的，但是没有原则性的方法来判断哪些是真实的。

随着科学的进步，我们的理论越来越接近现实。至少有一些假设的实体是真实的。

**SR3:** 有些理论是完全正确的，有些则是近似正确的。真实理论假定的实体，以及近似理论中的一些实体，都是真实的。

**SR4:** 如果一个理论正确地描述了现实，那么它就是正确的，否则就是错误的。真正的理论所假定的实体是真实的，其他的则不是。

**SR4** 是如此强大，以至于它可能是站不住脚的；根据这样一个严格的标准，几乎所有当前的理论都是错误的。大多数现实主义者会接受介于 **SR1** 和 **SR3** 之间的东西。

## 6.5 工具主义

但 **SR1** 是如此的脆弱，以至于它接近于工具主义，工具主义认为理论是我们用于我们目的的工具：一个理论是否有用，取决于它是否符合它的目的，但是我们不能说它是否正确。

为了看看你是否适应工具主义，我做了以下测试。阅读下面的陈述，并为每一个你认同的陈述给自己一个观点。如果你得到 4 分或更多，你可能是一个乐器演奏家！

“Entities in the Game of Life aren’t real; they are just patterns of cells that people have given cute names.”

“A hurricane is just a pattern of air flow, but it is a useful description because it allows us to make predictions and communicate about the weather.”

“Freudian entities like the Id and the Superego aren’t real, but they are useful tools for thinking and communicating about psychology (or at least some people think so).”

“Electric and magnetic fields are postulated entities in our best theory of electromagnetism, but they aren’t real. We could construct other theories, without postulating fields, that would be just as useful.”

“Many of the things in the world that we identify as objects are arbitrary collections like constellations. For example, a mushroom is just the fruiting body of a fungus, most of which grows underground as a barely-contiguous network of cells. We focus on mushrooms for practical reasons like visibility and edibility.”

“Some objects have sharp boundaries, but many are fuzzy. For example, which molecules are part of your body: Air in your lungs? Food in your stomach? Nutrients in your blood? Nutrients in a cell? Water in a cell? Structural parts of a cell? Hair? Dead skin? Dirt? Bacteria on your skin? Bacteria in your gut? Mitochondria? How many of those molecules do you include when you weigh yourself? Conceiving the world in terms of discrete objects is useful, but the entities we identify are not real.”

If you are more comfortable with some of these statements than others, ask yourself why. What are the differences in these scenarios that influence your reaction? Can you make a principled distinction between them?

For more on instrumentalism, see <http://thinkcomplex.com/instr>.

生命游戏中的实体并不真实，它们只是人们给予可爱名字的细胞模式。”

飓风只是空气流动的一种模式，但它是一种有用的描述，因为它使我们能够预测和交流有关天气的信息。”

像本我和超我这样的弗洛伊德实体并不真实，但它们是思考和交流心理学的有用工具(至少有些人是这么认为的)。”

在我们最好的电磁学理论中，电场和磁场是假设的实体，但它们不是真实的。我们可以构建其它理论，而不需要假设，这也同样有用。”

世界上很多我们认为是对象的东西都是任意的集合，比如星座。例如，蘑菇只是真菌的子实体，大多数真菌生长在地下，形成一个几乎不连续的细胞网络。我们关注蘑菇是出于可见性和可食性等实际原因。”

有些物体有明显的边界，但许多物体是模糊的。例如，哪些分子是你身体的一部分：你肺里的空气？胃里的食物？你血液中的营养物质？细胞中的营养物质？牢房里的水？细胞的结构部分？头发？死皮？泥土？皮肤上的细菌？肠道中的细菌？线粒体？当你称体重的时候，你包含了多少这样的分子？用离散的对象来构想世界是有用的，但我们识别的实体是不真实的。”

如果你比其他人更能接受这些陈述，问问自己为什么。在这些情况下，哪些因素影响了你的反应？你能对他们进行有原则的区分吗？

更多关于工具主义的信息，请参阅《<http://thinkcomplex.com/instr>》。

## 6.6 Implementing Life

The exercises at the end of this chapter ask you to experiment with and modify the Game of Life, and implement other 2-D cellular automata. This section explains my implementation of GoL, which you can use as a starting place for your experiments.

To represent the state of the cells, I use a NumPy array of 8-bit unsigned integers. As an example, the following line creates a 10 by 10 array initialized with random values of 0 and 1.

```
a = np.random.randint(2, size=(10, 10), dtype=np.uint8)
```

There are a few ways we can compute the GoL rules. The simplest is to use `for` loops to iterate through the rows and columns of the array:

```
b = np.zeros_like(a)
rows, cols = a.shape
for i in range(1, rows-1):
    for j in range(1, cols-1):
        state = a[i, j]
        neighbors = a[i-1:i+2, j-1:j+2]
        k = np.sum(neighbors) - state
        if state:
            if k==2 or k==3:
                b[i, j] = 1
        else:
            if k == 3:
                b[i, j] = 1
```

Initially, `b` is an array of zeros with the same size as `a`. Each time through the loop, `state` is the condition of the center cell and `neighbors` is the 3x3 neighborhood. `k` is the number of live neighbors (not including the center cell). The nested `if` statements evaluate the GoL rules and turn on cells in `b` accordingly.

This implementation is a straightforward translation of the rules, but it is verbose and slow. We can do better using cross-correlation, as we saw in Section 5.11. There, we used `np.correlate` to compute a 1-D correlation.

## 6.6 实现生活

本章最后的练习要求你们实验和修改生命的游戏，并实现其他 2-D 元胞自动机。这一节说明我对 GoL 的实现，你可以把它作为你实验的起点。

为了表示单元格的状态，我使用了一个 8 位无符号整数的 NumPy 数组。作为一个示例，下面的行创建一个初始化为 0 和 1 的随机值的  $10 \times 10$  数组。

```
A = np.random.randint(2, size=(10,10), dtype=np.uint8)
```

有几种方法可以计算 GoL 规则。最简单的方法是循环迭代数组的行和列：

```
0_like(a)
行, cols = a 形状
I 在范围(1, 行-1):
    为 j in range(1, cols-1):
        状态 = a[i, j]
        邻居 = a[i-1:i+2, j-1:j+2]
        K = np.sum(neighbors)-state
        如果状态:
            如果 k = 2 或者 k = 3:
                B[i, j] = 1
        其他:
            如果 k = 3:
                B[i, j] = 1
```

最初，`b` 是一个与 `a` 大小相同的零数组。每次通过循环时，状态是中心单元的状态，邻居是  $3 \times 3$  的邻居。`K` 是活的邻居的数量(不包括中间的细胞)。嵌套的 `if` 语句计算 GoL 规则并相应地打开 `b` 中的单元格。

此实现是对规则的直接转换，但是它冗长而缓慢。正如我们在 5.11 节中看到的，我们可以更好地使用互相关。在那里，我们使用 `np.correlate` 来计算一维相关性。

Now, to perform 2-D correlation, we'll use `correlate2d` from `scipy.signal`, a SciPy module that provides functions related to signal processing:

```
from scipy.signal import correlate2d

kernel = np.array([[1, 1, 1],
                  [1, 0, 1],
                  [1, 1, 1]])

c = correlate2d(a, kernel, mode='same')
```

What we called a “window” in the context of 1-D correlation is called a “kernel” in the context of 2-D correlation, but the idea is the same: `correlate2d` multiplies the kernel and the array to select a neighborhood, then adds up the result. This kernel selects the 8 neighbors that surround the center cell.

`correlate2d` applies the kernel to each location in the array. With `mode='same'`, the result has the same size as `a`.

Now we can use logical operators to compute the rules:

```
b = (c==3) | (c==2) & a
b = b.astype(np.uint8)
```

The first line computes a boolean array with `True` where there should be a live cell and `False` elsewhere. Then `astype` converts the boolean array to an array of integers.

This version is faster, and probably good enough, but we can simplify it slightly by modifying the kernel:

```
kernel = np.array([[1, 1, 1],
                  [1,10, 1],
                  [1, 1, 1]])

c = correlate2d(a, kernel, mode='same')
b = (c==3) | (c==12) | (c==13)
b = b.astype(np.uint8)
```

This version of the kernel includes the center cell and gives it a weight of 10. If the center cell is 0, the result is between 0 and 8; if the center cell is 1,

现在，为了执行二维关联，我们将使用来自 `SciPy.signal` 的 `correlate2d`，`SciPy` 模块提供与信号处理相关的功能：

```
来自 scipy.signal import correlate2d

Kernel = np.array ([1,1,1],
                   [1,0,1],
                   [1,1,1])

C = correlate2d (a, kernel, mode = ' same')
```

我们在一维相关性上下文中所谓的窗口在二维相关性上下文中称为一个核，但其思想是一样的：相关二维将核和数组相乘以选择一个邻域，然后将结果相加。这个内核选择中心单元周围的 8 个邻居。

`Correlate2d` 将内核应用到数组中的每个位置。对于 `mode = " same"`，结果与 `a` 的大小相同。

现在我们可以使用逻辑运算符来计算规则：

```
B = (c = 3) | (c = 2) & a
B = b.astype (np.uint8)
```

第一行用 `True` 计算一个布尔数组，其中应该有活动单元格，而其他地方应该有 `False`。然后，`astype` 将布尔数组转换为整数数组。

这个版本更快，可能也足够好了，但是我们可以通过修改内核来简化它：

```
Kernel = np.array ([1,1,1],
                   [1,10,1],
                   [1,1,1])

C = correlate2d (a, kernel, mode = ' same') b = (c
= 3) | (c = 12) | (c = 13)
B = b.astype (np.uint8)
```

这个版本的内核包含了中心单元，并赋予它 10 的重量。如果中心单元格为 0，则结果介于 0 和 8 之间；如果中心单元格为 1，

the result is between 10 and 18. Using this kernel, we can simplify the logical operations, selecting only cells with the values 3, 12, and 13.

That might not seem like a big improvement, but it allows one more simplification: with this kernel, we can use a table to look up cell values, as we did in Section 5.12.

```
table = np.zeros(20, dtype=np.uint8)
table[[3, 12, 13]] = 1
c = correlate2d(a, kernel, mode='same')
b = table[c]
```

`table` has zeros everywhere except locations 3, 12, and 13. When we use `c` as an index into `table`, NumPy performs element-wise lookup; that is, it takes each value from `c`, looks it up in `table`, and puts the result into `b`.

This version is faster and more concise than the others; the only drawback is that it takes more explaining.

`Life.py`, which is included in the repository for this book, provides a `Life` class that encapsulates this implementation of the rules. If you run `Life.py`, you should see an animation of a “puffer train”, which is a spaceship that leaves a trail of detritus in its wake.

## 6.7 Exercises

The code for this chapter is in the Jupyter notebook `chap06.ipynb` in the repository for this book. Open this notebook, read the code, and run the cells. You can use this notebook to work on the following exercises. My solutions are in `chap06soln.ipynb`.

**Exercise 6.1** Start GoL in a random state and run it until it stabilizes. What stable patterns can you identify?

**Exercise 6.2** Many named patterns are available in portable file formats. Modify `Life.py` to parse one of these formats and initialize the grid.

**Exercise 6.3** One of the longest-lived small patterns is “rabbits”, which starts with 9 live cells and takes 17,331 steps to stabilize. You can get the initial

结果是 10 到 18 岁之间。使用这个内核，我们可以简化逻辑操作，只选择值为 3、12 和 13 的单元格。

这可能看起来不是一个很大的改进，但是它允许一个更简单的改进：使用这个内核，我们可以使用一个表来查找单元格值，就像我们在 5.12 节中所做的那样。

```
Table = np.zeros(20, dtype = np.uint8)
table[3,12,13] = 1
c = correlate2d(a, kernel, mode = 'same')
b = table[c]
```

除了位置 3、12 和 13 之外，表的每个地方都有 0。当我们使用 `c` 作为表的索引时，NumPy 执行元素查找；也就是说，它从 `c` 中获取每个值，在表中查找它，并将结果放入 `b` 中。

这个版本比其他版本更快，更简洁；唯一的缺点是需要更多的解释。

Py 包含在本书的存储库中，它提供了一个 `Life` 类来封装这些规则的实现。如果你运行 `Life.py`，你应该看到一个普尔火车的动画”，这是一个太空船在它的尾部留下碎屑的痕迹。

## 6.7 练习

本章的代码在本书资料库中的 `chapyter` 笔记本 `chap06.ipynb` 中。打开这个笔记本，阅读代码，并运行单元格。你可以用这个笔记本做以下练习。我的解决方案在第 6 章 `soln.ipynb`。

练习 6.1 以随机状态开始 `GoL`，并运行它直到稳定下来。你能识别出什么稳定的模式？

练习 6.2 许多命名的模式都可以以便携的 `le` 格式提供。

修改 `Life.py` 以解析其中一种格式并初始化网格。

运动 6.3 最长寿的小模式之一是“兔子”，它从 9 个活细胞开始，经过 17,331 步才能稳定下来。你可以得到首字母

configuration in various formats from <http://thinkcomplex.com/rabbits>. Load this configuration and run it.

**Exercise 6.4** In my implementation, the `Life` class is based on a parent class called `Cell2D`, and the `LifeViewer` class is based on `Cell2DViewer`. You can use these base classes to implement other 2-D cellular automata.

For example, one variation of GoL, called “Highlife”, has the same rules as GoL, plus one additional rule: a dead cell with 6 neighbors comes to life.

Write a class named `Highlife` that inherits from `Cell2D` and implements this version of the rules. Also write a class named `HighlifeViewer` that inherits from `Cell2DViewer` and try different ways to visualize the results. As a simple example, use a different colormap.

One of the more interesting patterns in Highlife is the replicator (see <http://thinkcomplex.com/repl>). Use `add_cells` to initialize Highlife with a replicator and see what it does.

**Exercise 6.5** If you generalize the Turing machine to two dimensions, or add a read-write head to a 2-D CA, the result is a cellular automaton called a Turmite. It is named after a termite because of the way the read-write head moves, but spelled wrong as an homage to Alan Turing.

The most famous Turmite is Langton’s Ant, discovered by Chris Langton in 1986. See <http://thinkcomplex.com/langton>.

The ant is a read-write head with four states, which you can think of as facing north, south, east or west. The cells have two states, black and white.

The rules are simple. During each time step, the ant checks the color of the cell it is on. If black, the ant turns to the right, changes the cell to white, and moves forward one space. If the cell is white, the ant turns left, changes the cell to black, and moves forward.

Given a simple world, a simple set of rules, and only one moving part, you might expect to see simple behavior — but you should know better by now. Starting with all white cells, Langton’s ant moves in a seemingly random pattern for more than 10,000 steps before it enters a cycle with a period of 104 steps. After each cycle, the ant is translated diagonally, so it leaves a trail called the “highway”.

不同形式的 <http://thinkcomplex.com/rabbits>。

载入这个饱和度并运行它。

练习 6.4 在我的实现中，`Life` 类基于一个名为 `Cell2D` 的父类，`LifeViewer` 类基于 `Cell2DViewer`。您可以使用这些基类来实现其他 2-D 元胞自动机。

例如，GoL 的一个变体叫做“高生命”，它有着与 GoL 相同的规则，另外还有一个规则：一个有 6 个邻居的死亡细胞复活。

编写一个名为 `Highlife` 的类，它继承自 `Cell2D` 并实现这个版本的规则。还要编写一个名为 `HighlifeViewer` 的类，该类继承自 `Cell2DViewer`，并尝试不同的方法来可视化结果。作为一个简单的例子，使用一个不同的彩色图。

`Highlife` 中一个比较有趣的模式是 `replicator` (见 [http:// thinkcomplex.com/repl](http://thinkcomplex.com/repl))。使用 `add_cells` 用一个复制器初始化 `Highlife`，然后看看它会做什么。

练习 6.5 如果你把图灵机推广到二维空间，或者在二维 CA 中增加一个读写头，结果就是一个叫做细胞自动机的图灵。它是以一只白蚁的名字命名的，因为它的读写头移动的方式，但拼写错误是为了向阿兰·图灵致敬。

最著名的是 1986 年 Chris Langton 发现的 Langton 蚂蚁，参见 <http://thinkcomplex.com/Langton>。

蚂蚁是一个有四个状态的读写头，你可以认为它们面向北、南、东或西。细胞有两种状态，黑色和白色。

规则很简单。在每一个时间步骤中，蚂蚁都会检查所处细胞的颜色。如果是黑色，蚂蚁向右转，将细胞变成白色，并向前移动一个空间。如果细胞是白色的，蚂蚁向左转，把细胞变成黑色，然后向前移动。

给定一个简单的世界，一组简单的规则，只有一个移动的部分，你可能期望看到简单的行为 | 但是现在你应该知道得更清楚了。从所有的白细胞开始，兰顿的蚂蚁以一种看似随机的模式移动了超过 10000 步，然后进入了一个周期为 104 步的循环。在每个循环之后，蚂蚁都会沿着对角线移动，所以它会留下一条被称为“高速公路”的痕迹。

Write an implementation of Langton's Ant.

写一个 Langton's Ant 的实现。





# Chapter 7

## Physical modeling

The cellular automata we have seen so far are not physical models; that is, they are not intended to describe systems in the real world. But some CAs are intended as physical models.

In this chapter we consider a CA that models chemicals that diffuse (spread out) and react with each other, which is a process Alan Turing proposed to explain how some animal patterns develop.

And we'll experiment with a CA that models percolation of liquid through porous material, like water through coffee grounds. This model is the first of several models that exhibit **phase change** behavior and **fractal geometry**, and I'll explain what both of those mean.

The code for this chapter is in `chap07.ipynb` in the repository for this book. More information about working with the code is in Section 0.3.

### 7.1 Diffusion

In 1952 Alan Turing published a paper called “The chemical basis of morphogenesis”, which describes the behavior of systems involving two chemicals that diffuse in space and react with each other. He showed that these systems produce a wide range of patterns, depending on the diffusion and reaction rates, and conjectured that systems like this might be an important mechanism in

## 第七章

### 物理模型

到目前为止，我们所看到的细胞自动机不是物理模型，也就是说，它们不是用来描述现实世界中的系统。但是有些 `ca` 是用作物理模型的。

在这一章中，我们考虑一个 `CA`，它模拟化学物质的分散和相互作用，这是阿兰·图灵提出的解释一些动物模式如何发展的过程。

我们将用一个 `CA` 进行实验，该 `CA` 模拟液体通过多孔材料的过滤过程，就像水通过地面一样。这个模型是几个表现出相变行为和分形几何的模型中的第一个，我将解释这两者的含义。

本章的代码位于本书知识库中的 `chap07.ipynb` 中。

关于使用代码的更多信息请参见 0.3 部分。

#### 7.1 Discussion

1952 年，阿兰·图灵发表了一篇名为形态发生的化学基础的论文，描述了涉及两种化学物质的系统的行为，这两种化学物质在空间中互相使用并发生反应。他指出，这些系统产生广泛的模式，取决于分散和反应速率，并推测这样的系统可能是一个重要的机制

biological growth processes, particularly the development of animal coloration patterns.

Turing's model is based on differential equations, but it can be implemented using a cellular automaton.

Before we get to Turing's model, we'll start with something simpler: a diffusion system with just one chemical. We'll use a 2-D CA where the state of each cell is a continuous quantity (usually between 0 and 1) that represents the concentration of the chemical.

We'll model the diffusion process by comparing each cell with the average of its neighbors. If the concentration of the center cell exceeds the neighborhood average, the chemical flows from the center to the neighbors. If the concentration of the center cell is lower, the chemical flows the other way.

The following kernel computes the difference between each cell and the average of its neighbors:

```
kernel = np.array([[0, 1, 0],
                  [1,-4, 1],
                  [0, 1, 0]])
```

Using `np.correlate2d`, we can apply this kernel to each cell in an array:

```
c = correlate2d(array, kernel, mode='same')
```

We'll use a diffusion constant, `r`, that relates the difference in concentration to the rate of flow:

```
array += r * c
```

Figure 7.1 shows results for a CA with size `n=9`, diffusion constant `r=0.1`, and initial concentration 0 everywhere except for an "island" in the middle. The figure shows the starting configuration and the state of the CA after 5 and 10 steps. The chemical spreads from the center outward, continuing until the concentration is the same everywhere.

生物生长过程，特别是动物着色模式的发展。

图灵的模型是基于微分方程的，但是它可以用细胞自动机来实现。

在我们进入图灵的模型之前，我们先从一些简单的东西开始：一个只有一种化学物质的分散系统。我们将使用一个二维 CA，其中每个细胞的状态是一个连续的量(通常介于 0 和 1 之间)，代表化学品的浓度。

我们将通过比较每个细胞与其邻近细胞的平均值来模拟分化过程。如果中心细胞的浓度超过邻近细胞的平均浓度，化学浆就会从中心流向邻近细胞。如果中心细胞的浓度较低，化学物质的浓度则相反。

下面的内核计算每个细胞与其邻居的平均值之间的差异：

```
Kernel = np.array ([0,1,0],  
                   [1,-4,1]  
                   [0,1,0])
```

使用 `np.correlate2d`，我们可以将这个内核应用到数组中的每个单元格：

```
C = correlate2d (array, kernel, mode = ' same')
```

我们将使用一个 diffusion 常数，`r`，它将浓度的差值与低的速率联系起来：

```
数组 += r * c
```

图 7.1 显示了一个大小为 `n = 9`，直流常数 `r = 0.1`，初始浓度为 0 的 CA 的结果，除了中间的一个岛。图中显示了起始饱和度和经过 5 步和 10 步后的 CA 状态。化学物质从中心向外扩散，继续扩散，直到各处浓度相同。

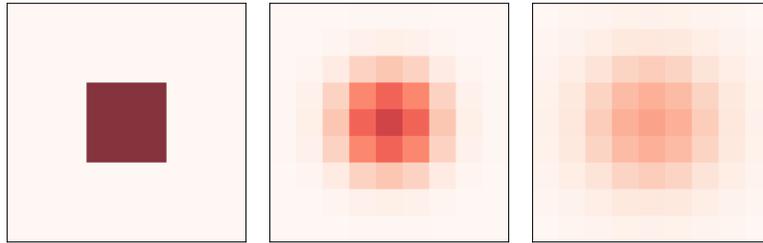


Figure 7.1: A simple diffusion model after 0, 5, and 10 steps.

## 7.2 Reaction-diffusion

Now let's add a second chemical. I'll define a new object, `ReactionDiffusion`, that contains two arrays, one for each chemical:

```
class ReactionDiffusion(Cell2D):  
  
    def __init__(self, n, m, params, noise=0.1):  
        self.params = params  
        self.array = np.ones((n, m), dtype=float)  
        self.array2 = noise * np.random.random((n, m))  
        add_island(self.array2)
```

`n` and `m` are the number of rows and columns in the array. `params` is a tuple of parameters, which I explain below.

`array` represents the concentration of the first chemical, `A`; the NumPy function `ones` initializes it to 1 everywhere. The data type `float` indicates that the elements of `A` are floating-point values.

`array2` represents the concentration of `B`, which is initialized with random values between 0 and `noise`, which is 0.1 by default. Then `add_island` adds an island of higher concentration in the middle:

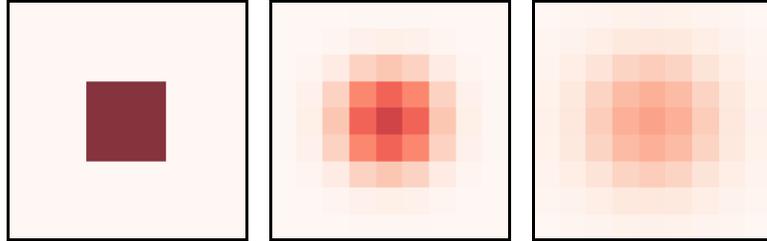


图 7.1: 经过 0、5 和 10 个步骤后的简单分析模型。

## 7.2 反应-结论

现在让我们加入第二种化学物质。我将创建一个新对象，`ReactionDiffusion`，它包含两个数组，每个数组代表一种化学物质：

类反应扩散(Cell2D)：

0.1)：

```
1. params = params
```

```
Array = np.ones ((n, m) , dtype = float)
```

```
2 = noise * np.random.random ((n, m))
```

```
Add _ island (self.array2)
```

`N` 和 `m` 是数组中的行数和列数。`Params` 是一组参数，我将在下面解释。

数组表示第一个化学物质 `a` 的浓度；NumPy 函数的值将其初始化为 1。数据类型 `float` 表示 `a` 的元素是定点值。

数组 2 表示 `b` 的浓度，初始化值在 0 和噪声之间，默认为 0.1。然后在中间加上一个高密度的岛屿：

```
def add_island(a, height=0.1):
    n, m = a.shape
    radius = min(n, m) // 20
    i = n//2
    j = m//2
    a[i-radius:i+radius, j-radius:j+radius] += height
```

The radius of the island is one twentieth of  $n$  or  $m$ , whichever is smaller. The height of the island is `height`, with the default value 0.1.

Here is the `step` function that updates the arrays:

```
def step(self):
    A = self.array
    B = self.array2
    ra, rb, f, k = self.params

    cA = correlate2d(A, self.kernel, **self.options)
    cB = correlate2d(B, self.kernel, **self.options)

    reaction = A * B**2
    self.array += ra * cA - reaction + f * (1-A)
    self.array2 += rb * cB + reaction - (f+k) * B
```

The parameters are

The diffusion rate of A (analogous to  $r$  in the previous section).

The diffusion rate of B. In most versions of this model,  $rb$  is about half of  $ra$ .

The “feed” rate, which controls how quickly A is added to the system.

The “kill” rate, which controls how quickly B is removed from the system.

Now let’s look more closely at the update statements:

```
reaction = A * B**2
self.array += ra * cA - reaction + f * (1-A)
self.array2 += rb * cB + reaction - (f+k) * B
```



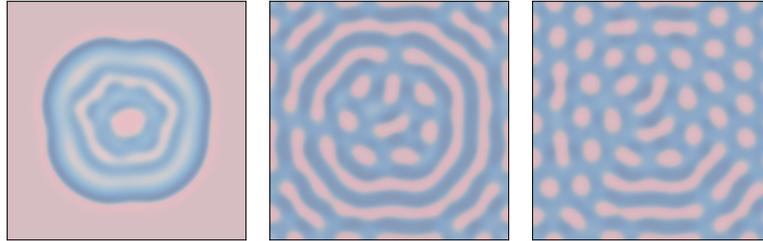


Figure 7.2: Reaction-diffusion model with parameters  $f=0.035$  and  $k=0.057$  after 1000, 2000, and 4000 steps.

The arrays  $cA$  and  $cB$  are the result of applying a diffusion kernel to  $A$  and  $B$ . Multiplying by  $ra$  and  $rb$  yields the rate of diffusion into or out of each cell.

The term  $A * B**2$  represents the rate that  $A$  and  $B$  react with each other. Assuming that the reaction consumes  $A$  and produces  $B$ , we subtract this term in the first equation and add it in the second.

The term  $f * (1-A)$  determines the rate that  $A$  is added to the system. Where  $A$  is near 0, the maximum feed rate is  $f$ . Where  $A$  approaches 1, the feed rate drops off to zero.

Finally, the term  $(f+k) * B$  determines the rate that  $B$  is removed from the system. As  $B$  approaches 0, this rate goes to zero.

As long as the rate parameters are not too high, the values of  $A$  and  $B$  usually stay between 0 and 1.

With different parameters, this model can produce patterns similar to the stripes and spots on a variety of animals. In some cases, the similarity is striking, especially when the feed and kill parameters vary in space.

For all simulations in this section,  $ra=0.5$  and  $rb=0.25$ .

Figure 7.2 shows results with  $f=0.035$  and  $k=0.057$ , with the concentration of  $B$  shown in darker colors. With these parameters, the system evolves toward a stable configuration with light spots of  $A$  on a dark background of  $B$ .

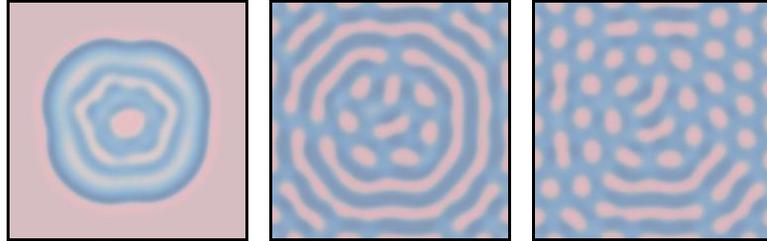


图 7.2: 参数  $f = 0.035$  和  $k = 0.057$  的反应-扩散模型经过 1000, 2000 和 4000 步后。

数组  $cA$  和  $cB$  是对  $a$  和  $b$  应用分离核的结果。  
乘以  $ra$  和  $rb$  得出每个细胞的挤入或挤出速率。

术语  $a * b * * 2$  表示  $a$  和  $b$  相互作用的速率。假设反应消耗  $a$  生成  $b$ ，我们在第一个方程中减去这个项，在第二个方程中加上它。

术语  $f * (1-A)$  决定  $a$  加入系统的速率。当  $a$  接近 0 时，最大进给速度为  $f$ ；当  $a$  接近 1 时，进给速度降为 0 到零。

最后，项  $(f + k) * b$  确定  $b$  从系统中移除的速率。当  $b$  接近 0 时，这个比率变为 0。

只要速率参数不太高， $a$  和  $b$  的值通常保持在 0 到 1 之间。

通过不同的参数，该模型可以产生类似于各种动物身上的条纹和斑点的图案。在某些情况下，相似性是惊人的，特别是当提供和压井参数在空间上不同时。

对于本节中的所有模拟， $ra = 0.5$  和  $rb = 0.25$ 。

图 7.2 显示了  $f = 0.035$  和  $k = 0.057$  的结果， $b$  的浓度显示为深色。在这些参数的作用下，系统逐渐向着稳定的饱和过程发展，在  $b$  的深色背景上出现亮点  $a$ 。

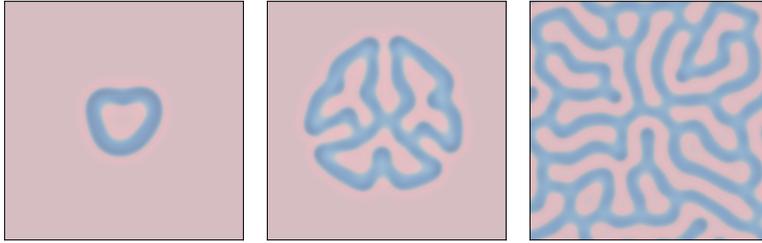


Figure 7.3: Reaction-diffusion model with parameters  $f=0.055$  and  $k=0.062$  after 1000, 2000, and 4000 steps.

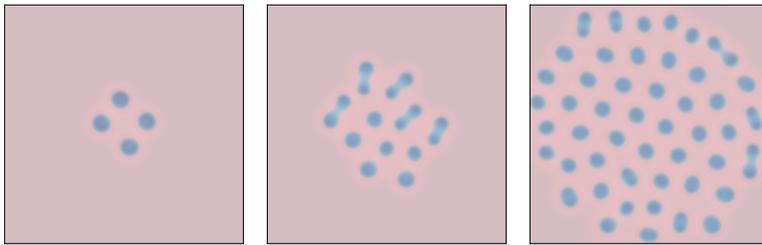


Figure 7.4: A reaction-diffusion model with parameters  $f=0.039$  and  $k=0.065$  after 1000, 2000, and 4000 steps.

Figure 7.3 shows results with  $f=0.055$  and  $k=0.062$ , which yields a coral-like pattern of B on a background of A.

Figure 7.4 shows results with  $f=0.039$  and  $k=0.065$ . These parameters produce spots of B that grow and divide in a process that resembles mitosis, ending with a stable pattern of equally-spaced spots.

Since 1952, observations and experiments have provided some support for Turing's conjecture. At this point it seems likely, but not yet proven, that many animal patterns are actually formed by reaction-diffusion processes of some kind.

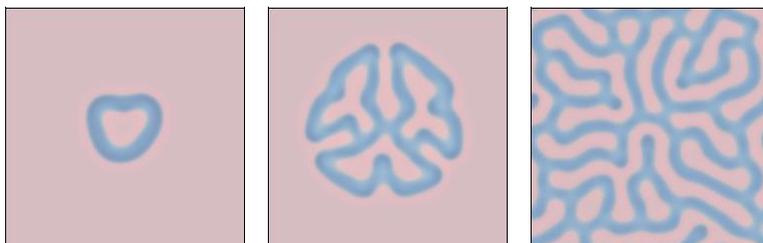


图 7.3: 参数  $f = 0.055$  和  $k = 0.062$  的反应-扩散模型经过 1000,2000 和 4000 步后。

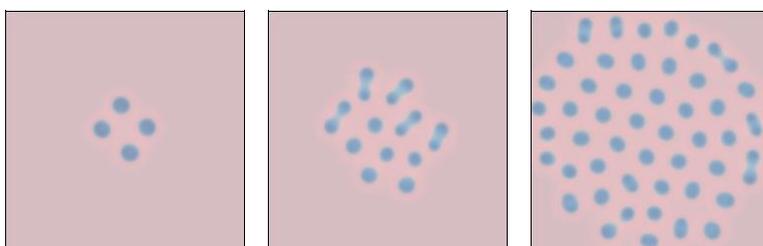


图 7.4: 参数  $f = 0.039$  和  $k = 0.065$  的反应-扩散模型经过 1000,2000 和 4000 步后。

图 7.3 显示了  $f = 0.055$  和  $k = 0.062$  的结果，它在  $a$  的背景上产生了一个珊瑚状的  $b$  图案。

图 7.4 显示了  $f = 0.039$  和  $k = 0.065$  的结果。这些参数产生的  $b$  点在类似于有丝分裂的过程中生长和分裂，最终形成稳定的等距斑点模式。

自 1952 年以来，观测和实验为图灵猜想提供了一些支持。在这一点上，似乎有可能，但尚未被证实，许多动物模式实际上是由某种反应-挤压过程形成的。

## 7.3 Percolation

Percolation is a process in which a fluid flows through a semi-porous material. Examples include oil in rock formations, water in paper, and hydrogen gas in micropores. Percolation models are also used to study systems that are not literally percolation, including epidemics and networks of electrical resistors. See <http://thinkcomplex.com/perc>.

Percolation models are often represented using random graphs like the ones we saw in Chapter 2, but they can also be represented using cellular automata. In the next few sections we'll explore a 2-D CA that simulates percolation.

In this model:

- Initially, each cell is either “porous” with probability  $q$  or “non-porous” with probability  $1-q$ .
- When the simulation begins, all cells are considered “dry” except the top row, which is “wet”.
- During each time step, if a porous cell has at least one wet neighbor, it becomes wet. Non-porous cells stay dry.
- The simulation runs until it reaches a “fixed point” where no more cells change state.

If there is a path of wet cells from the top to the bottom row, we say that the CA has a “percolating cluster”.

Two questions of interest regarding percolation are (1) the probability that a random array contains a percolating cluster, and (2) how that probability depends on  $q$ . These questions might remind you of Section 2.3, where we considered the probability that a random Erdős-Rényi graph is connected. We will see several connections between that model and this one.

I define a new class to represent a percolation model:

```
class Percolation(Cell2D):  
  
    def __init__(self, n, q):  
        self.q = q  
        self.array = np.random.choice([1, 0], (n, n), p=[q, 1-q])  
        self.array[0] = 1
```

## 7.3 过滤

渗流是一种半多孔材料渗透的过程。例子包括岩层中的石油，纸中的水，微孔中的氢气。逾渗模型也被用于研究并非真正意义上的逾渗系统，包括流行病和电阻网络。参见 <http://thinkcomplex.com/perc>。

逾渗模型通常使用随机图表示，就像我们在第 2 章中看到的那样，但是它们也可以使用元胞自动机来表示。在接下来的几节中，我们将探索一个模拟渗滤的二维 CA。

在这个模型中：

- \* 最初，每个单元格要么是有孔隙的，概率为  $q$ ，要么是无孔隙的，概率为  $1-q$ 。

当模拟开始时，所有的电池都被认为是干的，除了上面一排是湿的。

在每个时间步骤中，如果一个多孔电池至少有一个潮湿的邻居，则该多孔电池将被称为多孔电池

变得潮湿，无孔电池保持干燥。

这个模拟一直运行，直到它到达一个折叠点，在那里没有更多的细胞改变状态。

如果存在一条湿细胞从上到下的路径，我们就说 CA 具有一个“渗滤簇”。

关于逾渗的两个有趣的问题是：(1)随机数组包含逾渗集群的概率，以及(2)这个概率如何依赖于  $q$ 。这些问题可能会让你想起 2.3 节，其中我们考虑了一个随机 Erdős-Rényi 图是连通的概率。我们将看到这个模型和这个模型之间的一些联系。

我需要一个新的类别来代表一个过滤模型：

课堂渗透(Cell2D)：

返回文章页面【一分钟科普】：

`= q`

`Self.array = np.random.choice ([1,0], (n, n), p = [ q, 1-q ])`

`[0] = 5`

$n$  and  $m$  are the number of rows and columns in the CA.

The state of the CA is stored in `array`, which is initialized using `np.random.choice` to choose 1 (porous) with probability  $q$ , and 0 (non-porous) with probability  $1-q$ .

The state of the top row is set to 5, which represents a wet cell. Using 5, rather than the more obvious 2, makes it possible to use `correlate2d` to check whether any porous cell has a wet neighbor. Here is the kernel:

```
kernel = np.array([[0, 1, 0],
                  [1, 0, 1],
                  [0, 1, 0]])
```

This kernel defines a 4-cell “von Neumann” neighborhood; unlike the Moore neighborhood we saw in Section 6.1, it does not include the diagonals.

This kernel adds up the states of the neighbors. If any of them are wet, the result will exceed 5. Otherwise the maximum result is 4 (if all neighbors happen to be porous).

We can use this logic to write a simple, fast `step` function:

```
def step(self):
    a = self.array
    c = correlate2d(a, self.kernel, mode='same')
    self.array[(a==1) & (c>=5)] = 5
```

This function identifies porous cells, where `a==1`, that have at least one wet neighbor, where `c>=5`, and sets their state to 5, which indicates that they are wet.

Figure 7.5 shows the first few steps of a percolation model with  $n=10$  and  $p=0.7$ . Non-porous cells are white, porous cells are lightly shaded, and wet cells are dark.

## 7.4 Phase change

Now let’s test whether a random array contains a percolating cluster:



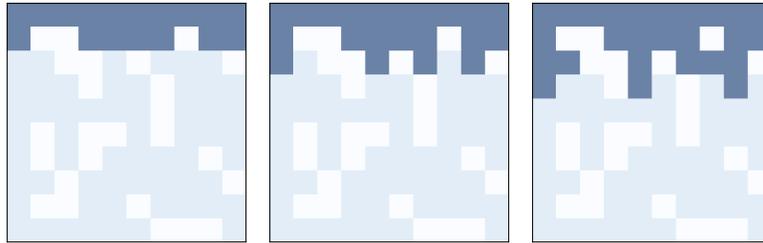


Figure 7.5: The first three steps of a percolation model with  $n=10$  and  $p=0.7$ .

```
def test_perc(perc):
    num_wet = perc.num_wet()

    while True:
        perc.step()

        if perc.bottom_row_wet():
            return True

        new_num_wet = perc.num_wet()
        if new_num_wet == num_wet:
            return False

        num_wet = new_num_wet
```

`test_perc` takes a `Percolation` object as a parameter. Each time through the loop, it advances the CA one time step. It checks the bottom row to see if any cells are wet; if so, it returns `True`, to indicate that there is a percolating cluster.

During each time step, it also computes the number of wet cells and checks whether the number increased since the last step. If not, we have reached a fixed point without finding a percolating cluster, so `test_perc` returns `False`.

To estimate the probability of a percolating cluster, we generate many random arrays and test them:

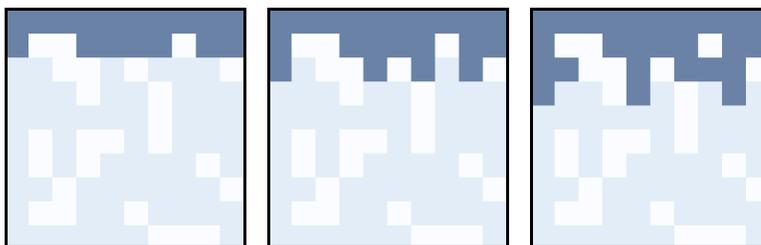


图 7.5:  $n = 10$  和  $p = 0.7$  的逾渗模型的第一个三个步骤。

```
Deftest_perc (perc) :
```

```
    Num_wet = perc.num _ wet ()
```

```
    虽然是真的:
```

```
        Perc.step ()
```

```
    如果 perc.bottom _ row _ wet () :
```

```
        返回 True
```

```
    新 num _ wet = perc.num _ wet ()
```

```
    如果 new num wet = num wet:
```

```
        返回 False
```

#### 2.5.5.5

使用一个 `Percolation` 对象作为参数。每次通过循环，它都将 CA 前进一个时间步。它检查底部的行，看看是否有任何单元格是湿的；如果有，它返回 `True`，以表明存在一个渗滤集群。

在每个时间步骤中，它还计算湿电池的数量，并检查自上一步以来是否有所增加。如果没有，那么我们已经到达了一个中止点，而没有发现一个过滤集群，因此 `test _ perc` 返回 `False`。

为了估计渗流集群的概率，我们生成了许多随机阵列并对它们进行了测试：

```
def estimate_prob_percolating(n=100, q=0.5, iters=100):
    t = [test_perc(Percolation(n, q)) for i in range(iters)]
    return np.mean(t)
```

`estimate_prob_percolating` makes 100 `Percolation` objects with the given values of `n` and `q` and calls `test_perc` to see how many of them have a percolating cluster. The return value is the fraction that do.

When  $p=0.55$ , the probability of a percolating cluster is near 0. At  $p=0.60$ , it is about 70%, and at  $p=0.65$  it is near 1. This rapid transition suggests that there is a critical value of  $p$  near 0.6.

We can estimate the critical value more precisely using a **random walk**. Starting from an initial value of `q`, we construct a `Percolation` object and check whether it has a percolating cluster. If so, `q` is probably too high, so we decrease it. If not, `q` is probably too low, so we increase it.

Here's the code:

```
def find_critical(n=100, q=0.6, iters=100):
    qs = [q]
    for i in range(iters):
        perc = Percolation(n, q)
        if test_perc(perc):
            q -= 0.005
        else:
            q += 0.005
        qs.append(q)
    return qs
```

The result is a list of values for `q`. We can estimate the critical value, `q_crit`, by computing the mean of this list. With `n=100` the mean of `qs` is about 0.59; this value does not seem to depend on `n`.

The rapid change in behavior near the critical value is called a **phase change** by analogy with phase changes in physical systems, like the way water changes from liquid to solid at its freezing point.

A wide variety of systems display a common set of behaviors and characteristics when they are at or near a critical point. These behaviors are known collectively as **critical phenomena**. In the next section, we explore one of them: fractal geometry.



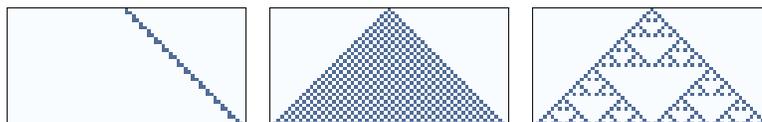


Figure 7.6: One-dimensional CAs with rules 20, 50, and 18, after 32 time steps.

## 7.5 Fractals

To understand fractals, we have to start with dimensions.

For simple geometric objects, dimension is defined in terms of scaling behavior. For example, if the side of a square has length  $l$ , its area is  $l^2$ . The exponent, 2, indicates that a square is two-dimensional. Similarly, if the side of a cube has length  $l$ , its volume is  $l^3$ , which indicates that a cube is three-dimensional.

More generally, we can estimate the dimension of an object by measuring some kind of size (like area or volume) as a function of some kind of linear measure (like the length of a side).

As an example, I'll estimate the dimension of a 1-D cellular automaton by measuring its area (total number of “on” cells) as a function of the number of rows.

Figure 7.6 shows three 1-D CAs like the ones we saw in Section 5.2. Rule 20 (left) generates a set of cells that seems like a line, so we expect it to be one-dimensional. Rule 50 (center) produces something like a triangle, so we expect it to be 2-D. Rule 18 (right) also produces something like a triangle, but the density is not uniform, so its scaling behavior is not obvious.

I'll estimate the dimension of these CAs with the following function, which counts the number of on cells after each time step. It returns a list of tuples, where each tuple contains  $i$ ,  $i^2$ , and the total number of cells.

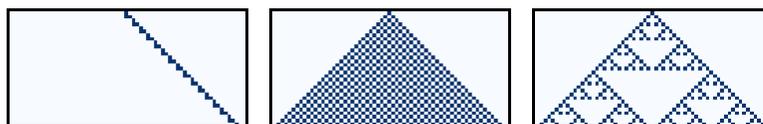


图 7.6: 经过 32 个时间步骤后, 具有规则 20、50 和 18 的一维 ca。

### 7.5 分形

为了解分形, 我们必须从维度开始。

对于简单的几何物体, 尺寸是根据缩放行为来确定的。例如, 如果一个正方形的边长为 1, 那么它的面积就是  $1^2$ 。指数 2 表示正方形是二维的。类似地, 如果立方体的边长为 1, 那么它的体积为  $1^3$ , 这表明立方体是三维的。

更一般地说, 我们可以通过测量某种尺寸(如面积或体积)作为某种线性测量(如边长)的函数来估计物体的尺寸。

作为一个例子, 我将通过测量单元格面积(单元格总数)与行数的函数来估计一维细胞自动机的大小。

图 7.6 显示了与我们在 5.2 节中看到的类似的三个一维 ca。规则 20(左)生成一组细胞, 看起来像一条线, 所以我们期望它是一维的。规则 50(中心)产生类似于三角形的东西, 所以我们希望它是 2-D 的。规则 18(右)也产生类似三角形的东西, 但密度不均匀, 因此其缩放行为不明显。

我将使用下面的函数来估计这些 ca 的维度, 该函数计算每个时间步骤之后单元格上的数量。它返回一个元组列表, 其中每个元组包含  $i$ 、 $i^2$  和单元格总数。

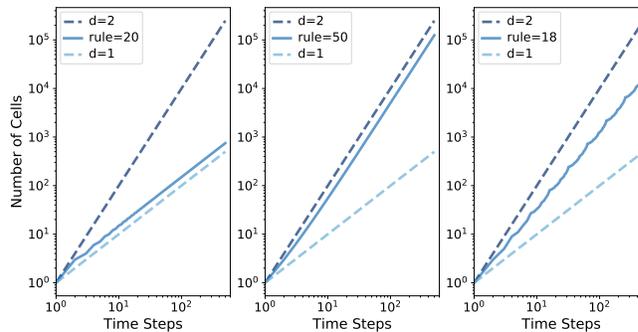


Figure 7.7: Number of “on” cells versus number of time steps for rules 20, 50, and 18.

```
def count_cells(rule, n=500):
    ca = Cell1D(rule, n)
    ca.start_single()

    res = []
    for i in range(1, n):
        cells = np.sum(ca.array)
        res.append((i, i**2, cells))
        ca.step()

    return res
```

Figure 7.7 shows the results plotted on a log-log scale.

In each figure, the top dashed line shows  $y = i^2$ . Taking the log of both sides, we have  $\log y = 2 \log i$ . Since the figure is on a log-log scale, the slope of this line is 2.

Similarly, the bottom dashed line shows  $y = i$ . On a log-log scale, the slope of this line is 1.

Rule 20 (left) produces 3 cells every 2 time steps, so the total number of cells after  $i$  steps is  $y = 1.5i$ . Taking the log of both sides, we have  $\log y = \log 1.5 + \log i$ , so on a log-log scale, we expect a line with slope 1. In fact, the estimated slope of the line is 1.01.

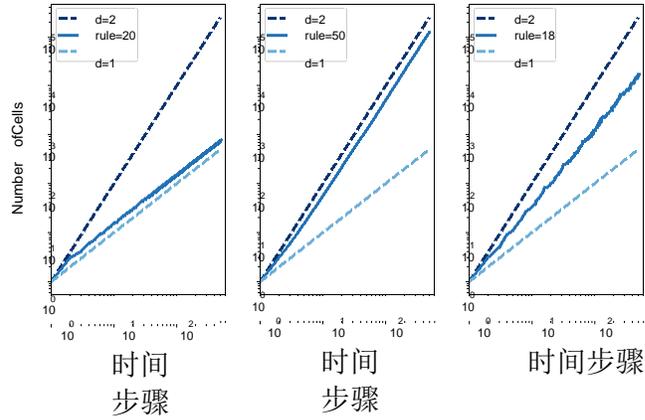


图 7.7: “on”单元格的数量与规则 20、50 和 18 的时间步骤的数量。

```
Def count_cells (rule, n = 500) :
```

```
    Ca = Cell1D (rule, n)
```

```
    Start_single ()
```

```
    Res = []
```

```
    I 在范围(1, n) :
```

```
        Cells = np.sum (ca.array)
```

```
        Res.append ((i, i ** 2, cells))
```

```
        Ca.step ()
```

```
    回报率
```

图 7.7 显示了在对数对数尺度上绘制的结果。

在每个图形中，上面的虚线表示  $y = i^2$ 。对两边的对数，我们有对数  $y = 2$  对数  $i$ 。由于该曲线是对数曲线，因此该曲线的斜率为 2。

类似地，下面的虚线表示  $y = i$ 。在对数尺度上，这条线的斜率是 1。

规则 20(左)每 2 个时间步产生 3 个单元格，所以  $i$  步后的单元格总数为  $y = 1.5i$ 。事实上，这条线的斜率估计是 1.01。

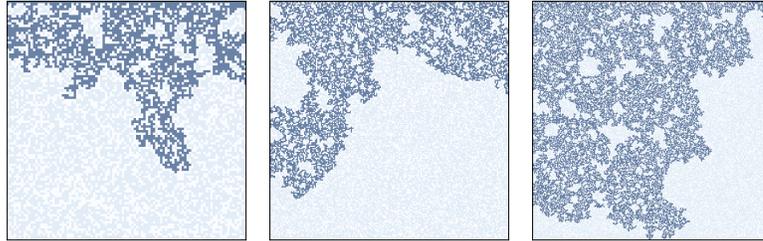


Figure 7.8: Percolation models with  $q=0.6$  and  $n=100, 200,$  and  $300$ .

Rule 50 (center) produces  $i + 1$  new cells during the  $i$ th time step, so the total number of cells after  $i$  steps is  $y = i^2 + i$ . If we ignore the second term and take the log of both sides, we have  $\log y \sim 2 \log i$ , so as  $i$  gets large, we expect to see a line with slope 2. In fact, the estimated slope is 1.97.

Finally, for Rule 18 (right), the estimated slope is about 1.57, which is clearly not 1, 2, or any other integer. This suggests that the pattern generated by Rule 18 has a “fractional dimension”; that is, it is a fractal.

This way of estimating a fractal dimension is called **box-counting**. For more about it, see <http://thinkcomplex.com/box>.

## 7.6 Fractals and Percolation Models

Now let’s get back to percolation models. Figure 7.8 shows clusters of wet cells in percolation simulations with  $p=0.6$  and  $n=100, 200,$  and  $300$ . Informally, they resemble fractal patterns seen in nature and in mathematical models.

To estimate their fractal dimension, we can run CAs with a range of sizes, count the number of wet cells in each percolating cluster, and then see how the cell counts scale as we increase the size of the array.

The following loop runs the simulations:

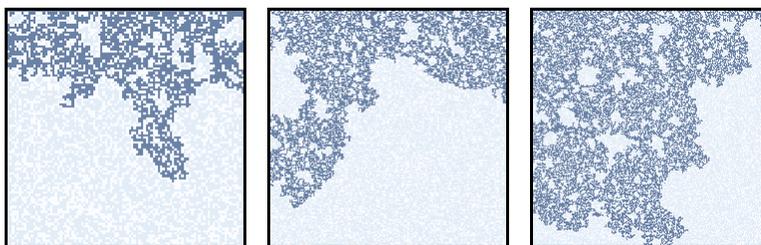


图 7.8:  $q = 0.6$  和  $n = 100, 200$  和  $300$  的逾渗模型。

规则 50(中心)在第 1 个时间步骤中产生  $i + 1$  个新单元, 所以在  $i$  步骤之后的单元总数为  $y = i^2 + i$ 。如果我们忽略第二项, 取两边的对数, 我们有对数  $y \approx 2 \log i$ , 所以当  $i$  变大时, 我们希望看到斜率为 2 的直线。事实上, 估计的斜率是 1.97。

最后, 对于规则 18(右), 估计的斜率约为 1.57, 这显然不是 1、2 或任何其他整数。这表明, 规则 18 生成的模式具有分形维数”, 也就是说, 它是一个分形。

这种估计分形维数的方法被称为盒子计数法, 关于它的更多信息, 请参阅《<http://thinkcomplex.com/box>》。

### 7.6 分形和渗流模型

现在让我们回到过滤模型。图 7.8 显示了渗滤模拟中  $p = 0.6$  和  $n = 100, 200$  和  $300$  的湿细胞群。它们非正式地类似于自然界和数学模型中的分形图案。

为了估计它们的分形维数, 我们可以运行一系列大小的  $ca$ , 计算每个渗透集群中的湿细胞数量, 然后看看当我们增加阵列大小时细胞计数如何变化。

下面的循环运行模拟:

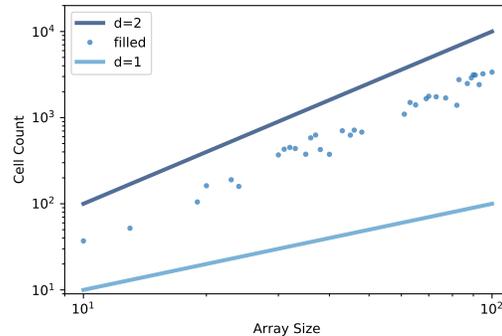


Figure 7.9: Number of cells in the percolating cluster versus CA size.

```

res = []
for size in sizes:
    perc = Percolation(size, q)
    if test_perc(perc):
        num_filled = perc.num_wet() - size
        res.append((size, size**2, num_filled))

```

The result is a list of tuples where each tuple contains `size`, `size**2`, and the number of cells in the percolating cluster (not including the initial wet cells in the top row).

Figure 7.9 shows the results for a range of sizes from 10 to 100. The dots show the number of cells in each percolating cluster. The slope of a line fitted to these dots is often near 1.85, which suggests that the percolating cluster is, in fact, fractal when  $q$  is near the critical value.

When  $q$  is larger than the critical value, nearly every porous cell gets filled, so the number of wet cells is close to  $q * \text{size}^2$ , which has dimension 2.

When  $q$  is substantially smaller than the critical value, the number of wet cells is proportional to the linear size of the array, so it has dimension 1.

## 7.7 Exercises

**Exercise 7.1** In Section 7.6 we showed that the Rule 18 CA produces a fractal. Can you find other 1-D CAs that produce fractals?

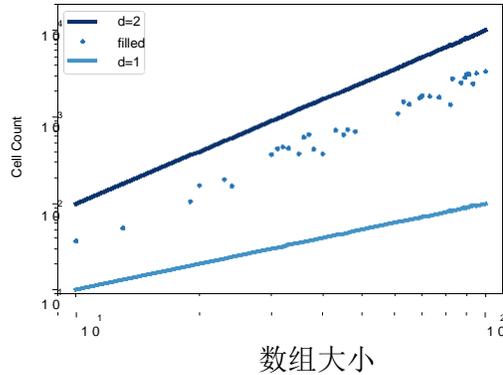


图 7.9: 渗滤集群中的细胞数量与 CA 大小的关系。

```
Res = []
```

对于尺寸大小:

```
Perc = Percolation (size, q)
```

如果 test\_perc (perc):

```
Num_filled = perc.num_wet ()-size res.append
(size, size ** 2, num_filled)
```

结果是一个元组列表，其中每个元组包含 size、size \*\* 2 和 percolating cluster 中的单元格数量(不包括顶行中的初始湿单元格)。

图 7.9 显示了从 10 到 100 的大小范围的结果。这些点显示了每个渗滤集群中的细胞数量。与这些点形成的直线的斜率通常接近 1.85，这表明当 q 值接近临界值时，渗滤集群实际上是分形的。

当 q 大于临界值时，几乎每个多孔单元都被拉长，因此湿单元的数量接近 q \* 尺寸 ^ 2，其维数为 2。

当 q 基本上小于临界值时，湿单元的数目与阵列的线性尺寸成正比，因此它的维数为 1。

## 7.7 练习

练习 7.1 在 7.6 节中，我们展示了规则 18 CA 产生了一个分形。你能找到其他产生分形的一维 ca 吗？

Note: the `Cell1D` object does not wrap around from the left edge to the right, which creates artifacts at the boundaries for some rules. You might want to use `Wrap1D`, which is a child class of `Cell1D` that wraps around. It is defined in `Cell1D.py` in the repository for this book.

**Exercise 7.2** In 1990 Bak, Chen and Tang proposed a cellular automaton that is an abstract model of a forest fire. Each cell is in one of three states: empty, occupied by a tree, or on fire.

The rules of the CA are:

1. An empty cell becomes occupied with probability  $p$ .
2. A cell with a tree burns if any of its neighbors is on fire.
3. A cell with a tree spontaneously burns, with probability  $f$ , even if none of its neighbors is on fire.
4. A cell with a burning tree becomes an empty cell in the next time step.

Write a program that implements this model. You might want to inherit from `Cell1D`. Typical values for the parameters are  $p = 0.01$  and  $f = 0.001$ , but you might want to experiment with other values.

Starting from a random initial condition, run the model until it reaches a steady state where the number of trees no longer increases or decreases consistently.

In steady state, is the geometry of the forest fractal? What is its fractal dimension?

注意: `Cell1D` 对象不会从左边缘绕到右边缘, 这会在某些规则的边界上创建工作。您可能需要使用 `Wrap1D`, 它是 `Cell1D` 的一个子类, 环绕在周围。本书储存在 `Cell1D.py` 中。

作业 7.21990 年 Bak, Chen 和 Tang 提出了一个细胞自动机, 这是一个森林重建的抽象模型。每个细胞都处于三种状态之一: 空的, 被树占据的, 或者在重复的状态。

通讯局的规则如下:

1. 一个空单元被  $p$  的概率占据。
2. 一个有树的细胞, 如果它的任何一个邻居在燃烧, 就会燃烧。
3. 一个有树的细胞自发燃烧, 概率为  $f$ , 即使它的邻居没有一个在燃烧。
4. 具有燃烧树的单元格在下一个时间步骤中成为空单元格。

编写一个实现这个模型的程序。你可能想从 `Cell2D` 继承。参数的典型值是  $p = 0.01$  和  $f = 0.001$ , 但是您可能希望使用其他值进行实验。

从一个随机的初始条件开始, 运行模型, 直到它达到一个稳定的状态, 树的数量不再增加或减少一致。

在稳定状态下, 森林的几何形态是分形的吗? 它的分形维数是什么?





# Chapter 8

## Self-organized criticality

In the previous chapter we saw an example of a system with a critical point and we explored one of the common properties of critical systems, fractal geometry.

In this chapter, we explore two other properties of critical systems: heavy-tailed distributions, which we saw in Chapter 4.4 and pink noise, which I'll explain in this chapter.

These properties are interesting in part because they appear frequently in nature; that is, many natural systems produce fractal-like geometry, heavy-tailed distributions, and pink noise.

This observation raises a natural question: why do so many natural systems have properties of critical systems? A possible answer is **self-organized criticality** (SOC), which is the tendency of some systems to evolve toward, and stay in, a critical state.

In this chapter I'll present a **sand pile model** that was the first system shown to exhibit SOC.

The code for this chapter is in `chap08.ipynb` in the repository for this book. More information about working with the code is in Section 0.3.

### 8.1 Critical Systems

Many critical systems demonstrate common behaviors:

## 第八章

### 自组织临界性

在前一章中，我们看到了一个具有临界点的系统的例子，我们探讨了临界系统的一个共同性质，分形几何。

在本章中，我们将探讨关键系统的另外两个性质：重尾分布，我们在第 4.4 章中已经看到，以及粉红噪声，我将在本章中解释。

这些特性之所以有趣，部分是因为它们在自然界中频繁出现；也就是说，许多自然系统会产生分形几何、重尾分布和粉红噪声。

这个观察结果提出了一个自然的问题：为什么这么多的自然系统具有关键系统的特性？一个可能的答案是自组织临界性，这是一些系统朝着临界状态进化并保持的趋势。

在这一章中，我将提出一个沙堆模型，这是第一个系统展示 SOC。

本章的代码位于本书知识库中的 `chap08.ipynb` 中。

关于使用代码的更多信息请参见 0.3 部分。

### 8.1 关键系统

许多关键系统展示了常见的行为：

- Fractal geometry: For example, freezing water tends to form fractal patterns, including snowflakes and other crystal structures. Fractals are characterized by self-similarity; that is, parts of the pattern are similar to scaled copies of the whole.
- Heavy-tailed distributions of some physical quantities: For example, in freezing water the distribution of crystal sizes is characterized by a power law.
- Variations in time that exhibit **pink noise**: Complex signals can be decomposed into their frequency components. In pink noise, low-frequency components have more power than high-frequency components. Specifically, the power at frequency  $f$  is proportional to  $1/f$ .

Critical systems are usually unstable. For example, to keep water in a partially frozen state requires active control of the temperature. If the system is near the critical temperature, a small deviation tends to move the system into one phase or the other.

Many natural systems exhibit characteristic behaviors of criticality, but if critical points are unstable, they should not be common in nature. This is the puzzle Bak, Tang and Wiesenfeld address. Their solution is called self-organized criticality (SOC), where “self-organized” means that from any initial condition, the system moves toward a critical state, and stays there, without external control.

## 8.2 Sand Piles

The sand pile model was proposed by Bak, Tang and Wiesenfeld in 1987. It is not meant to be a realistic model of a sand pile, but rather an abstraction that models physical systems with a large number of elements that interact with their neighbors.

The sand pile model is a 2-D cellular automaton where the state of each cell represents the slope of a part of a sand pile. During each time step, each cell is checked to see whether it exceeds a critical value,  $K$ , which is usually 3. If so, it “topples” and transfers sand to four neighboring cells; that is, the slope of the cell is decreased by 4, and each of the neighbors is increased by 1. At

分形几何: 例如, 冰冻的水往往形成分形图案, 包括雪迹和其他晶体结构。分形是拥有属性的自相似性, 也就是说, 图案的某些部分是相似的比例复制整个。

某些物理量的重尾分布: 例如, 在冰冻水中, 晶体大小的分布是拥有属性法律。

出现粉红噪声的时间变化: 复杂信号可以分解成它们的频率分量。在粉红噪声中, 低频分量比高频分量具有更大的功率。特别地, 频率  $f$  的功率与  $1/f$  成正比。

临界系统通常是不稳定的。例如, 要使水保持部分冻结状态, 就需要主动控制温度。如果系统接近临界温度, 一个小的偏差倾向于使系统进入一个相或另一个相。

许多自然系统表现出临界性的特征行为, 但是如果临界点是不稳定的, 那么它们在自然界中就不应该是普遍的。这是贝、唐和维森菲尔德的拼图地址。他们的解决方案被称为自组织临界性, 在这里自组织意味着从任何初始条件, 系统移动到一个临界状态, 并停留在那里, 没有外部控制。

## 8.2 沙堆

沙堆模型是 Bak, Tang 和 Wiesenfeld 在 1987 年提出的。它并不是一个真实的沙堆模型, 而是一个抽象的概念, 用大量的元素与它们的邻居相互作用来建模物理系统。

沙堆模型是一个二维的细胞自动机, 其中每个单元的状态代表一部分沙堆的坡度。在每个时间步骤中, 检查每个单元格是否超过一个临界值  $k$ , 这个临界值通常是 3。如果是这样的话, 它会倾倒并把沙子转移到四个相邻的细胞中, 也就是说, 细胞的斜率减少了 4, 而每个相邻的细胞的斜率增加了 1。在

the perimeter of the grid, all cells are kept at slope 0, so the excess spills over the edge.

Bak, Tang and Wiesenfeld initialize all cells at a level greater than  $K$  and run the model until it stabilizes. Then they observe the effect of small perturbations: they choose a cell at random, increment its value by 1, and run the model again until it stabilizes.

For each perturbation, they measure  $T$ , the number of time steps the pile takes to stabilize, and  $S$ , the total number of cells that topple<sup>1</sup>.

Most of the time, dropping a single grain causes no cells to topple, so  $T=1$  and  $S=0$ . But occasionally a single grain can cause an **avalanche** that affects a substantial fraction of the grid. The distributions of  $T$  and  $S$  turn out to be heavy-tailed, which supports the claim that the system is in a critical state.

They conclude that the sand pile model exhibits “self-organized criticality”, which means that it evolves toward a critical state without the need for external control or what they call “fine tuning” of any parameters. And the model stays in a critical state as more grains are added.

In the next few sections I replicate their experiments and interpret the results.

## 8.3 Implementing the Sand Pile

To implement the sand pile model, I define a class called `SandPile` that inherits from `Cell2D`, which is defined in `Cell2D.py` in the repository for this book.

```
class SandPile(Cell2D):  
  
    def __init__(self, n, m, level=9):  
        self.array = np.ones((n, m)) * level
```

All values in the array are initialized to `level`, which is generally greater than the toppling threshold,  $K$ .

Here’s the `step` method that finds all cells above  $K$  and topples them:

---

<sup>1</sup>The original paper uses a different definition of  $S$ , but most later work uses this definition.

网格的周边，所有的细胞都保持在 0 的斜率上，所以多余的细胞溢出边缘。

Bak, Tang 和 Wiesenfeld 将所有细胞初始化到大于  $k$  的水平并运行模型直到它稳定下来。然后他们观察小扰动的影响：他们随机选择一个细胞，将其值增加 1，然后再运行模型直到它稳定下来。

对于每个扰动，他们测量  $t$ ，稳定堆所需的时间步数，和  $s$ ，倾覆 1 的细胞总数。

大多数情况下，丢弃一颗颗粒不会导致细胞倒塌，因此  $t = 1$  和  $s = 0$ 。但是有时候一个微粒就能引起雪崩，破坏一个相当大的晶格。 $T$  和  $s$  的分布为重尾分布，支持系统处于临界状态的说法。

他们得出结论，沙堆模型表现出自组织临界性，这意味着它进化到一个临界状态，而不需要外部控制或他们所谓的任何参数的内部调整。随着颗粒数的增加，模型处于临界状态。

在接下来的几节中，我将重复他们的实验并解释实验结果。

### 8.3 建造沙堆

为了实现沙堆模型，我创建了一个名为 `SandPile` 的类，它继承自 `Cell2D`，这个类在本书的仓库 `Cell2D.py` 中有详细介绍。

```
类沙堆(Cell2D):  
  
    9):  
        Self.array = np.ones ((n, m)) * 级
```

数组中的所有值都初始化为级，这通常大于倾斜阈值  $k$ 。

这里有一个步骤方法，把  $k$  以上的所有单元格都颠倒过来：

---

1 原文采用了  $s$  的直接定义，但后来的大多数作品都采用了  $s$  的直接定义。

```

kernel = np.array([[0, 1, 0],
                  [1,-4, 1],
                  [0, 1, 0]])

def step(self, K=3):
    toppling = self.array > K
    num_toppled = np.sum(toppling)
    c = correlate2d(toppling, self.kernel, mode='same')
    self.array += c
    return num_toppled

```

To show how `step` works, I'll start with a small pile that has two cells ready to topple:

```

pile = SandPile(n=3, m=5, level=0)
pile.array[1, 1] = 4
pile.array[1, 3] = 4

```

Initially, `pile.array` looks like this:

```

[[0 0 0 0 0]
 [0 4 0 4 0]
 [0 0 0 0 0]]

```

Now we can select the cells that are above the toppling threshold:

```

toppling = pile.array > K

```

The result is a boolean array, but we can use it as if it were an array of integers like this:

```

[[0 0 0 0 0]
 [0 1 0 1 0]
 [0 0 0 0 0]]

```

If we correlate this array with the kernel, it makes copies of the kernel at each location where `toppling` is 1.

```

c = correlate2d(toppling, kernel, mode='same')

```

And here's the result:



```
[[ 0  1  0  1  0]
 [ 1 -4  2 -4  1]
 [ 0  1  0  1  0]]
```

Notice that where the copies of the kernel overlap, they add up.

This array contains the change for each cell, which we use to update the original array:

```
pile.array += c
```

And here's the result.

```
[[0 1 0 1 0]
 [1 0 2 0 1]
 [0 1 0 1 0]]
```

So that's how `step` works.

With `mode='same'`, `correlate2d` considers the boundary of the array to be fixed at zero, so any grains of sand that go over the edge disappear.

`SandPile` also provides `run`, which calls `step` until no more cells topple:

```
def run(self):
    total = 0
    for i in itertools.count(1):
        num_toppled = self.step()
        total += num_toppled
        if num_toppled == 0:
            return i, total
```

The return value is a tuple that contains the number of time steps and the total number of cells that toppled.

If you are not familiar with `itertools.count`, it is an infinite generator that counts up from the given initial value, so the `for` loop runs until `step` returns 0. You can read about the `itertools` module at <http://thinkcomplex.com/iter>.

Finally, the `drop` method chooses a random cell and adds a grain of sand:

```
[01010]
 [ 1-42-41]
 [ 01010]
```

注意，当内核的副本重叠时，它们相加。

这个数组包含每个单元格的更改，我们用它来更新原始数组：

```
数组 += c
```

这就是结果。

```
这是一个很好的例子，你可以在这里找到你想要的
[10201]
[01010]
```

这就是步骤的工作原理。

对于 `mode = '相同'`，`correlate2d` 认为阵列的边界为零，所以任何超过边界的沙粒都会消失。

还提供了 `run`，调用 `step`，直到没有更多的细胞倒下：

```
Def run (self) :
    总计 = 0
    在 itertools.count (1) :
        Num _ flopped = self.step ()
        总共 += num 翻倒
    如果 num 翻倒 = 0:
        返回 i, 总数
```

返回值是一个元组，其中包含时间步骤的数目和被推翻的单元格的总数。

如果您不熟悉 `itertools.count`，那么它是一个从给定初始值计数的 `infinite` 生成器，因此 `for` 循环一直运行到步骤返回

0. 你可以在 <http://thinkcomplex.com/> 阅读 `itertools` 模块。

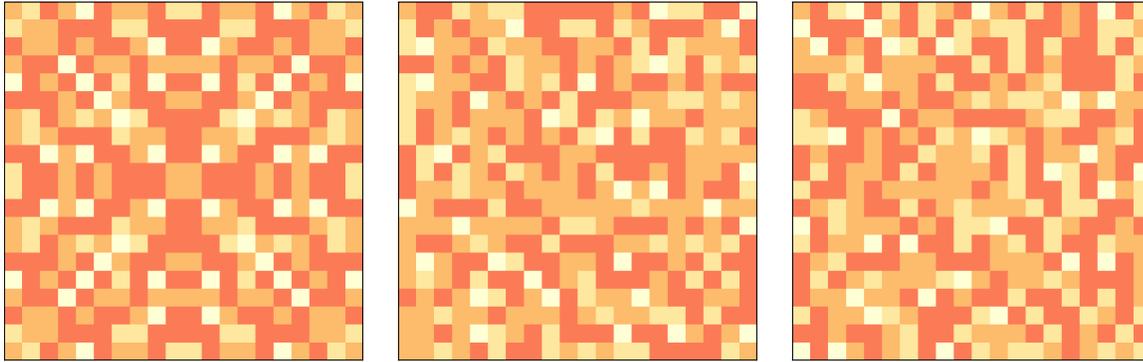


Figure 8.1: Sand pile model initial state (left), after 200 steps (middle), and 400 steps (right).

```
def drop(self):  
    a = self.array  
    n, m = a.shape  
    index = np.random.randint(n), np.random.randint(m)  
    a[index] += 1
```

Let's look at a bigger example, with  $n=20$ :

```
pile = SandPile(n=20, level=10)  
pile.run()
```

With an initial level of 10, this sand pile takes 332 time steps to reach equilibrium, with a total of 53,336 topplings. Figure 8.1 (left) shows the configuration after this initial run. Notice that it has the repeating elements that are characteristic of fractals. We'll come back to that soon.

Figure 8.1 (middle) shows the configuration of the sand pile after dropping 200 grains onto random cells, each time running until the pile reaches equilibrium. The symmetry of the initial configuration has been broken; the configuration looks random.

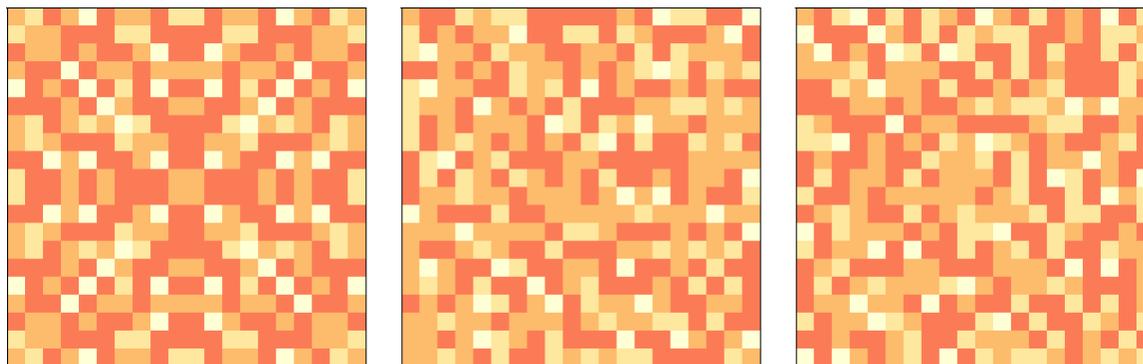


图 8.1: 沙堆模型的初始状态(左)，经过 200 个步骤(中)和 400 个步骤(右)。

```
Def drop (self) :  
    A = self.array  
    N, m = a.shape  
    Index = np.random.randint (n) , np.random.randint (m)  
    A [索引] += 1
```

让我们看一个更大的例子， $n = 20$ :

```
桩 = 沙堆(n = 20, 水平 = 10)  
Pile.run ()
```

初始水平为 10，需要 332 个时间步骤才能达到平衡，共计 53336 个顶点。图 8.1(左)显示了初始运行后的饱和度。请注意，它具有分形特征的重复元素。我们很快就会回到这个话题。

图 8.1(中间)显示了在随机单元上丢弃 200 颗沙粒后，沙堆的饱和度，每次直到桩达到平衡。初始饱和度的对称性被打破了，饱和度看起来是随机的。

Finally Figure 8.1 (right) shows the configuration after 400 drops. It looks similar to the configuration after 200 drops. In fact, the pile is now in a steady state where its statistical properties don't change over time. I'll explain some of those statistical properties in the next section.

## 8.4 Heavy-tailed distributions

If the sand pile model is in a critical state, we expect to find heavy-tailed distributions for quantities like the duration and size of avalanches. So let's take a look.

I'll make a larger sand pile, with `n=50` and an initial level of 30, and run until equilibrium:

```
pile2 = SandPile(n=50, level=30)
pile2.run()
```

Next, I'll run 100,000 random drops

```
iters = 100000
res = [pile2.drop_and_run() for _ in range(iters)]
```

As the name suggests, `drop_and_run` calls `drop` and `run` and returns the duration of the avalanche and total number of cells that toppled.

So `res` is a list of (T, S) tuples, where T is duration, in time steps, and S is cells toppled. We can use `np.transpose` to unpack `res` into two NumPy arrays:

```
T, S = np.transpose(res)
```

A large majority of drops have duration 1 and no toppled cells; if we filter them out before plotting, we get a clearer view of the rest of the distribution.

```
T = T[T>1]
S = S[S>0]
```

The distributions of T and S have many small values and a few very large ones. I'll use the `Pmf` class from `thinkstats2` to make a PMF of the values, that is, a map from each value to its probability of occurring (see Section 4.3).

最后，图 8.1(右)显示了 400 滴后的饱和度。它看起来类似于 200 滴后的饱和度。事实上，这一堆现在处于稳定状态，其统计特性不会随着时间的推移而改变。我将在下一节中解释其中的一些统计特性。

#### 8.4 重尾分布

如果沙堆模型处于临界状态，我们期望得到类似雪崩持续时间和规模的重尾分布。让我们来看看。

我会做一个更大的沙堆， $n = 50$ ，初始水平为 30，一直跑到平衡：

```
Pile2 = SandPile (n = 50, level = 30)
2. run ()
```

接下来，我会随机滴 10 万次

```
100000
Res = [ pile2.drop _ and _ run () for _ in range (iters)]
```

顾名思义，`drop _ and _ run` 调用删除并运行，返回雪崩的持续时间和崩溃的单元总数。

所以 `res` 是  $(t, s)$  元组的列表，其中  $t$  是持续时间，在时间步骤中， $s$  是单元格被推翻。我们可以使用 `np.transpose` 将 `res` 解压缩为两个 NumPy 数组：

```
T, s = np.transpose (res)
```

绝大多数的水滴持续时间为 1，没有倒塌的细胞；如果我们在策划之前把它们筛选出来，我们就能更清楚地了解其余的分布情况。

```
T = t [ t > 1]
S = s [ s > 0]
```

$T$  和  $s$  的分布有许多小值和一些非常大的值。我将使用 `thinkstats2` 中的 `PMF` 类来为这些值创建 `PMF`，即从每个值到其发生概率的映射(参见第 4.3 节)。

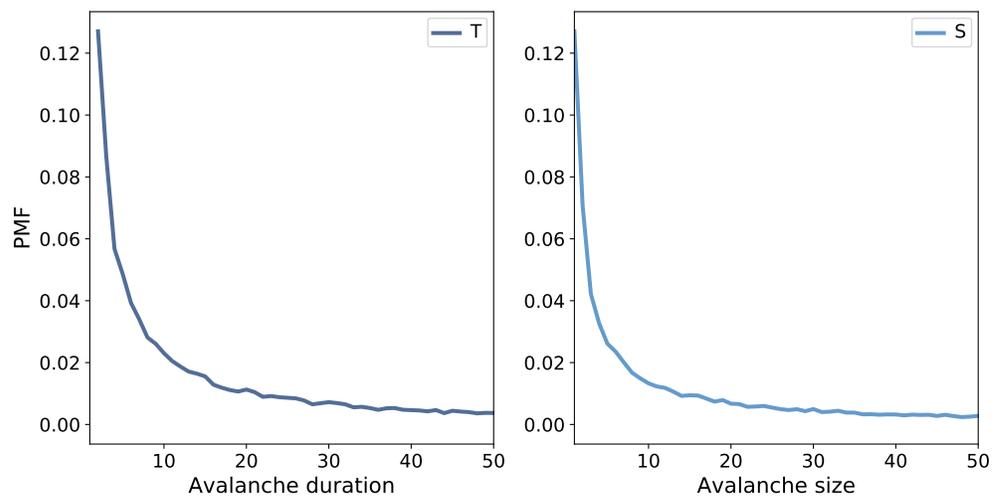


Figure 8.2: Distribution of avalanche duration (left) and size (right), linear scale.

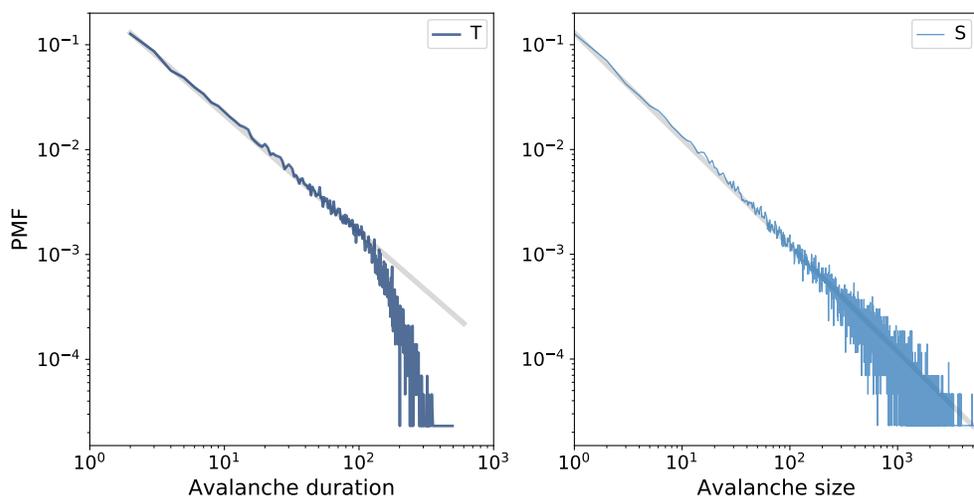


Figure 8.3: Distribution of avalanche duration (left) and size (right), log-log scale.

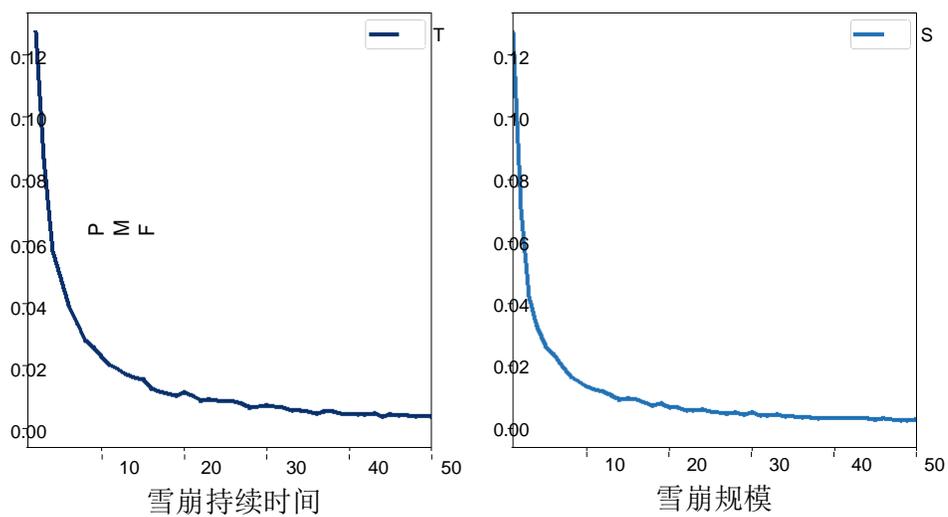


图 8.2: 雪崩持续时间的分布(左)和大小(右)，线性规模。

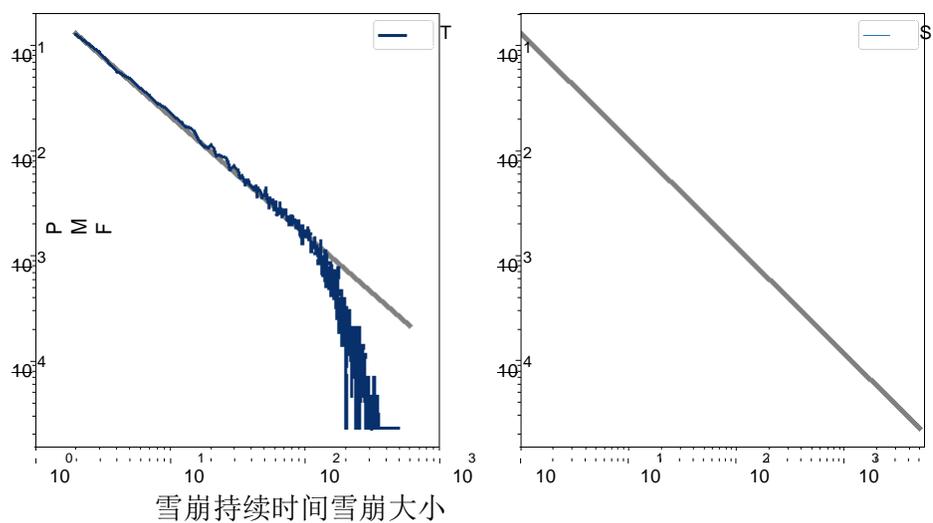


图 8.3: 雪崩持续时间的分布(左)和大小(右)，对数刻度。

```
pmfT = Pmf(T)
pmfS = Pmf(S)
```

Figure 8.2 shows the results for values less than 50.

As we saw in Section 4.4, we can get a clearer picture of these distributions by plotting them on a log-log scale, as shown in Figure 8.3.

For values between 1 and 100, the distributions are nearly straight on a log-log scale, which is characteristic of a heavy tail. The gray lines in the figure have slopes near -1, which suggests that these distributions follow a power law with parameters near  $\alpha = 1$ .

For values greater than 100, the distributions fall away more quickly than the power law model, which means there are fewer very large values than the model predicts. One possibility is that this effect is due to the finite size of the sand pile; if so, we might expect larger piles to fit the power law better.

Another possibility, which you can explore in one of the exercises at the end of this chapter, is that these distributions do not strictly obey a power law. But even if they are not power-law distributions, they might still be heavy-tailed.

## 8.5 Fractals

Another property of critical systems is fractal geometry. The initial configuration in Figure 8.1 (left) resembles a fractal, but you can't always tell by looking. A more reliable way to identify a fractal is to estimate its fractal dimension, as we saw in Section 7.5 and Section 7.6.

I'll start by making a bigger sand pile, with `n=131` and initial level 22.

```
pile3 = SandPile(n=131, level=22)
pile3.run()
```

It takes 28,379 steps for this pile to reach equilibrium, with more than 200 million cells toppled.

To see the resulting pattern more clearly, I select cells with levels 0, 1, 2, and 3, and plot them separately:

```
pmfT = Pmf (t)
pmfS = Pmf (s)
```

图 8.2 显示了小于 50 的结果。

正如我们在 4.4 节中看到的，我们可以通过在对数对数尺度上绘制这些分布，从而得到这些分布的更清晰的图像，如图 8.3 所示。

对于 1 到 100 之间的数值，分布在对数尺度上几乎是直线的，这是大尾巴的特征。图像中灰线的斜率接近-1，说明这些分布遵循参数接近 = 1 的幂律分布。

对于大于 100 的值，分布比幂律模型下降得更快，这意味着非常大的值比模型预测的要少。一种可能性是，这种影响是由于沙堆的大小决定的，如果是这样的话，我们可能会期望更大的沙堆更符合幂律。

另一种可能性，你可以在本章最后的练习中探讨，那就是这些分布并不严格遵守幂定律。但即使它们不是幂律分布，它们也可能是重尾分布。

## 8.5 分形

临界系统的另一个性质是分形几何。图 8.1(左)中的初始饱和度类似于分形，但是您不能总是通过观察来判断。识别一个分形更可靠的方法是估计它的分形维数，正如我们在第 7.5 节和第 7.6 节中看到的。

我先做一个更大的沙堆， $n = 131$ ，初始等级 22。

```
Pile3 = 沙堆(n = 131, level = 22)
3. run ()
```

这堆电池需要 28379 步才能达到平衡，有超过 2 亿个电池被推倒。

为了更清楚地看到结果模式，我选择了层级 0、1、2 和 3 的单元格，并分别绘制它们：

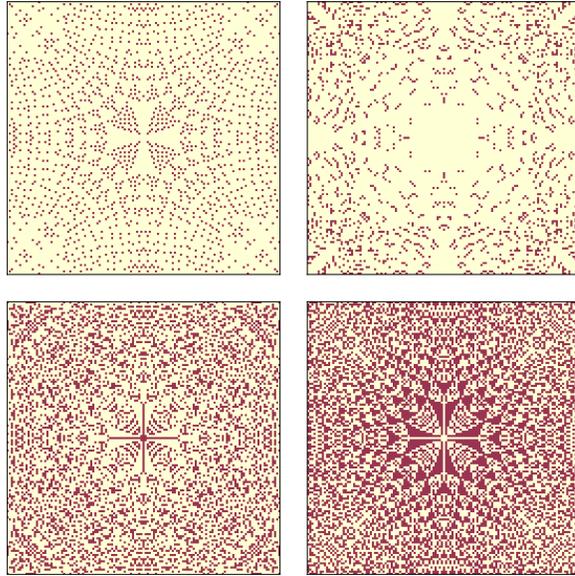


Figure 8.4: Sand pile model in equilibrium, selecting cells with levels 0, 1, 2, and 3, left to right, top to bottom.

```
def draw_four(viewer, levels=range(4)):
    thinkplot.preplot(rows=2, cols=2)
    a = viewer.viewee.array

    for i, level in enumerate(levels):
        thinkplot.subplot(i+1)
        viewer.draw_array(a==level, vmax=1)
```

`draw_four` takes a `SandPileViewer` object, which is defined in `Sand.py` in the repository for this book. The parameter `levels` is the list of levels we want to plot; the default is the range 0 through 3. If the sand pile has run until equilibrium, these are the only levels that should exist.

Inside the loop, it uses `a==level` to make a boolean array that's `True` where the array is `level` and `False` otherwise. `draw_array` treats these booleans as 1s and 0s.

Figure 8.4 shows the results for `pile3`. Visually, these patterns resemble fractals, but looks can be deceiving. To be more confident, we can estimate the fractal dimension for each pattern using **box-counting**, as we saw in Section 7.5.

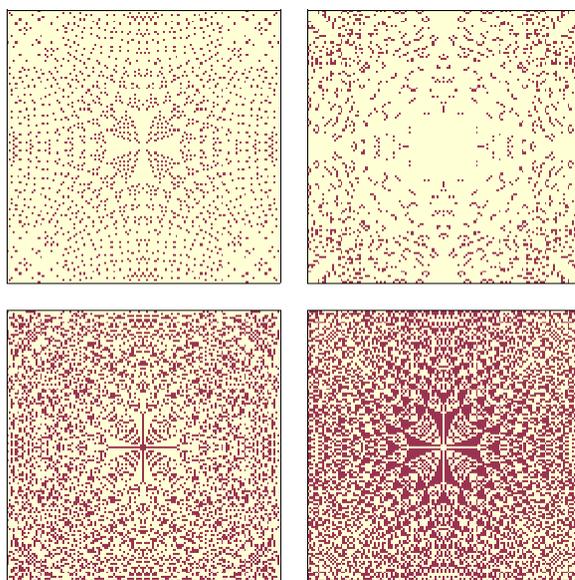


图 8.4: 沙堆模型处于平衡状态, 从左到右, 从上到下选择水平为 0、1、2 和 3 的单元格。

```

Def draw _ four (viewer, levels = range (4)) :
    thinkplot.preplot (rows = 2, cols = 2) a =
    viewer.viewee.array

```

对于  $i$ , 枚举级别(级别):

```
    Thinkplot.subplot (i + 1)
```

```
    Draw _ array (a == level, vmax = 1)
```

4 需要一个 `SandPileViewer` 对象, 这个对象在本书的仓库中被分解为 `Sand.py`。参数级别是我们要绘制的级别列表; 默认是从 0 到 3 的范围。如果沙堆一直运行到平衡, 这些是唯一的水平, 应该存在。

在循环中, 它使用 `== level` 创建一个布尔数组, 其中数组为 `level` 时为 `True`, 否则为 `False`。将这些布尔值分为 1 和 0。

图 8.4 显示了 `pile3` 的结果。从视觉上看, 这些模式类似于框架, 但外表可能具有欺骗性。为了更有说服力, 我们可以使用盒子计数来估计每个模式的分形维数, 正如我们在 7.5 节中看到的。

We'll count the number of cells in a small box at the center of the pile, then see how the number of cells increases as the box gets bigger. Here's my implementation:

```
def count_cells(a):
    n, m = a.shape
    end = min(n, m)

    res = []
    for i in range(1, end, 2):
        top = (n-i) // 2
        left = (m-i) // 2
        box = a[top:top+i, left:left+i]
        total = np.sum(box)
        res.append((i, i**2, total))

    return np.transpose(res)
```

The parameter, `a`, is a boolean array. The size of the box is initially 1. Each time through the loop, it increases by 2 until it reaches `end`, which is the smaller of `n` and `m`.

Each time through the loop, `box` is a set of cells with width and height `i`, centered in the array. `total` is the number of “on” cells in the box.

The result is a list of tuples, where each tuple contains `i`, `i**2`, and the number of cells in the box. When we pass this result to `transpose`, NumPy converts it to an array with three columns, and then **transposes** it; that is, it makes the columns into rows and the rows into columns. The result is an array with 3 rows: `i`, `i**2`, and `total`.

Here's how we use `count_cells`:

```
res = count_cells(pile.array==level)
steps, steps2, cells = res
```

The first line creates a boolean array that contains `True` where the array equals `level`, calls `count_cells`, and gets an array with three rows.

The second line unpacks the rows and assigns them to `steps`, `steps2`, and `cells`, which we can plot like this:

我们将计算堆中心一个小盒子中的单元格数量，然后看看随着盒子变大，单元格数量是如何增加的。下面是我的实现：

返回一个例子：

```
N, m = a.shape
End = min(n, m)
```

```
Res = []
对于 i 在范围(1, end, 2):
    Top = (n-i)//2
    左 = (m-i)//2
    Box = a [ top: top + i, left: left + i ]
    合计 = np.sum (方格)
    Res.append ((i i ** 2 total))
```

返回 np.transpose (res)

参数 `a` 是一个布尔数组。盒子的大小最初是 1。每次通过这个循环，它都会增加 2，直到它到达终点，也就是 `n` 和 `m` 的较小值。

每次通过循环时，`box` 都是一组单元格，其宽度和高度 `i` 以数组为中心。总数就是盒子里的“on”单元格的数目。

结果是一个元组列表，其中每个元组包含 `i`、`i ** 2` 和框中的单元格数。当我们将这个结果传递给 `transpose` 时，NumPy 将其转换为一个有三列的数组，然后将其转换；也就是说，它将列转换为行，将行转换为列。结果是一个包含 3 行的数组：`i`、`i ** 2` 和 `total`。

下面是我们使用 `count cell` 的方法：

```
Res = count _ cells (pile.array = level)
2, cells = res
```

第一行创建一个布尔数组，该数组包含 `True`，其中数组等于 `level`，调用 `count _ cells`，并获取一个具有三行的数组。

第二行展开这些行，并将它们分配给步骤、步骤 2 和单元格，我们可以这样绘制：

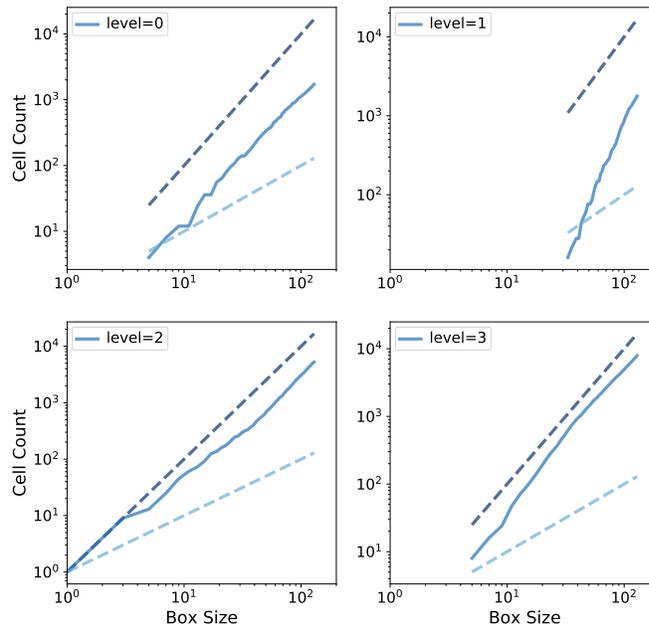


Figure 8.5: Box counts for cells with levels 0, 1, 2, and 3, compared to dashed lines with slopes 1 and 2.

```
thinkplot.plot(steps, steps2, linestyle='dashed')
thinkplot.plot(steps, cells)
thinkplot.plot(steps, steps, linestyle='dashed')
```

Figure 8.5 shows the results. On a log-log scale, the cell counts form nearly straight lines, which indicates that we are measuring fractal dimension over a valid range of box sizes.

To estimate the slopes of these lines, we can use the SciPy function `linregress`, which fits a line to the data by linear regression (see <http://thinkcomplex.com/regress>).

```
from scipy.stats import linregress

params = linregress(np.log(steps), np.log(cells))
slope = params[0]
```

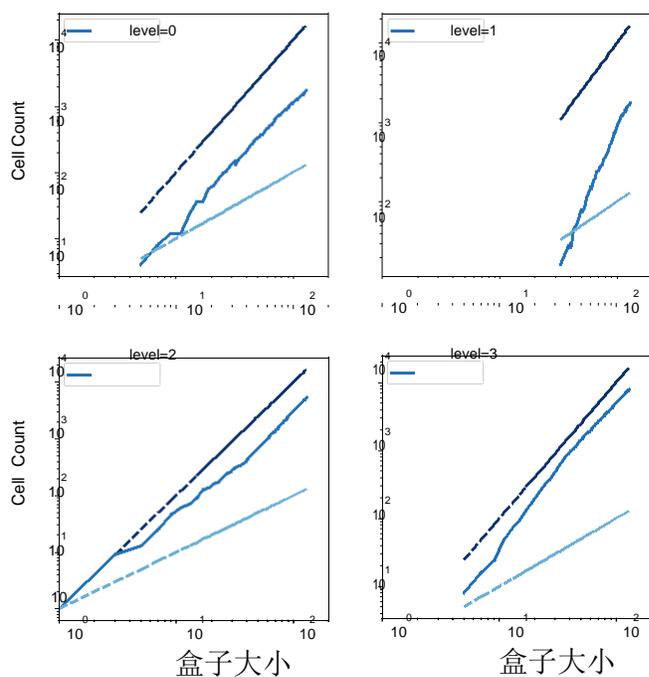


图 8.5: 水平为 0、1、2 和 3 的单元格计数，与斜率为 1 和 2 的虚线相比。

```
3. thinkplot.plot (steps, steps2, linestyle = '虚线')
Think plot.plot (steps, cells)
3. thinkplot.plot (steps, steps, linestyle = '虚线')
```

图 8.5 显示了结果。在对数对数尺度上，单元格计数几乎形成直线，这表明我们在有效的盒子尺寸范围内测量分形维数。

为了估计这些线的斜率，我们可以使用 SciPy 函数 `linregress`，它将一条线与线性回归的数据连接起来(参见 <http://thinkcomplex.Com/regress>)。)

```
来自 scipy.stats import linregress

Params = linregress (np.log (steps) , np.log (cells))
斜率 = 参数[0]
```

The estimated fractal dimensions are:

```
0  1.871
1  3.502
2  1.781
3  2.084
```

The fractal dimension for levels 0, 1, and 2 seems to be clearly non-integer, which indicates that the image is fractal.

The estimate for level 3 is indistinguishable from 2, but given the results for the other values, the apparent curvature of the line, and the appearance of the pattern, it seems likely that it is also fractal.

One of the exercises in the notebook for this chapter asks you to run this analysis again with different values of `n` and the initial `level` to see if the estimated dimensions are consistent.

## 8.6 Pink noise

The title of the original paper that presented the sand pile model is “Self-Organized Criticality: An Explanation of  $1/f$  Noise”. You can read it at <http://thinkcomplex.com/bak>.

As the subtitle suggests, Bak, Tang and Wiesenfeld were trying to explain why many natural and engineered systems exhibit  $1/f$  noise, which is also known as “flicker noise” and “pink noise”.

To understand pink noise, we have to take a detour to understand signals, power spectrums, and noise.

**Signal:** A **signal** is any quantity that varies in time. One example is sound, which is variation in air density. In the sand pile model, the signals we’ll consider are avalanche durations and sizes as they vary over time.

**Power spectrum:** Any signal can be decomposed into a set of frequency components with different levels of **power**, which is related to amplitude or volume. The **power spectrum** of a signal is a function that shows the power of each frequency component.

估计的分形维数如下:

```
0  1.871
   13.502
   21.781
   32.084
```

级别 0、1 和 2 的分形维数似乎是明显的非整数，这表明图像是分形的。

水平 3 的估计值与水平 2 无法区分，但是从其他值的结果来看，线的表观曲率和模式的外观来看，它似乎也是分形的。

本章笔记中的一个练习要求您用  $n$  的不同值和初始水平再次运行这个分析，看看估计的维度是否一致。

## 8.6 粉红噪音

提出沙堆模型的原始论文的标题是自组织临界性:  $1 = f$  噪声的解释。你可以在 <http://thinkcomplex.com/bak> 上阅读。

正如副标题所暗示的，**Bak**，**Tang** 和 **Wiesenfeld** 试图解释为什么许多自然和工程系统表现出  $1 = f$  噪音，也就是众所周知的踢脚噪音和粉红噪音。

为了解释粉红噪声，我们必须绕道去理解信号、功率谱和噪声。

**信号:** 信号是随时间变化的任何量。一个例子是声音，它是空气密度的变化。在沙堆模型中，我们将考虑的信号是雪崩持续时间和大小，因为它们随时间而变化。

**功率谱:** 任何信号都可以分解为一组不同功率水平的频率分量，这些频率分量与幅值或体积有关。信号的功率谱是显示每个频率分量功率的函数。

**Noise:** In common use, **noise** is usually an unwanted sound, but in the context of signal processing, it is a signal that contains many frequency components.

There are many kinds of noise. For example, “white noise” is a signal that has components with equal power over a wide range of frequencies.

Other kinds of noise have different relationships between frequency and power. In “red noise”, the power at frequency  $f$  is  $1/f^2$ , which we can write like this:

$$P(f) = 1/f^2$$

We can generalize this equation by replacing the exponent 2 with a parameter  $\beta$ :

$$P(f) = 1/f^\beta$$

When  $\beta = 0$ , this equation describes white noise; when  $\beta = 2$  it describes red noise. When the parameter is near 1, the result is called  $1/f$  noise. More generally, noise with any value between 0 and 2 is called “pink”, because it’s between white and red.

We can use this relationship to derive a test for pink noise. Taking the log of both sides yields

$$\log P(f) = -\beta \log f$$

So if we plot  $P(f)$  versus  $f$  on a log-log scale, we expect a straight line with slope  $-\beta$ .

What does this have to do with the sand pile model? Suppose that every time a cell topples, it makes a sound. If we record a sand pile model while its running, what would it sound like? In the next section, we’ll simulate the sound of the sand pile model and see if it is pink noise.

## 8.7 The sound of sand

As my implementation of `SandPile` runs, it records the number of cells that topple during each time step, accumulating the results in a list called `toppled_seq`. After running the model in Section 8.4, we can extract the resulting signal:

噪声: 在通常情况下, 噪声通常是一种不需要的声音, 但在信号处理的背景下, 它是一种包含许多频率成分的信号。

噪音有很多种。例如, “白噪声”是一种在很宽的频率范围内具有等功率分量的信号。

其它种类的噪声在频率和功率之间有着不同的关系。

在红噪声中, 频率  $f$  的功率是  $1 = f^2$ , 我们可以这样写:

$$P(f) = 1 = f^2$$

我们可以用一个参数代替指数 2 来推广这个方程

:

$$P(f) = 1 = f^\alpha$$

当  $\alpha = 0$  时, 这个方程描述了白噪声; 当  $\alpha = 2$  时, 它描述了红噪声。当参数接近 1 时, 结果被称为  $1 = f$  噪声。一般来说, 任何值在 0 到 2 之间的噪声都被称为“粉红色”, 因为它介于白色和红色之间。

我们可以利用这个关系导出一个粉红噪声的测试

$$\text{对数 } p(f) = \log f$$

所以如果我们在对数尺度上画  $p(f)$  和  $f$ , 我们期望斜率是一条直线。

这和沙堆模型有什么关系? 假设每次一个细胞倒下, 它就发出一个声音。如果我们在沙堆模型运行时录制它, 它听起来会是什么样子? 在下一节中, 我们将模拟沙堆模型的声音, 看看它是否是粉红噪声。

## 8.7 沙子的声音

当我运行 `SandPile` 的时候, 它会记录每个步骤中倒塌的细胞的数量, 并将结果累积在一个名为 `topt_seq` 的列表中。在第 8.4 节中运行模型之后, 我们可以提取结果信号:

```
signal = pile2.toppled_seq
```

To compute the power spectrum of this signal we can use the SciPy function `welch`:

```
from scipy.signal import welch

nperseg = 2048
freqs, spectrum = welch(signal, nperseg=nperseg, fs=nperseg)
```

This function uses Welch’s method, which splits the signal into segments and computes the power spectrum of each segment. The result is typically noisy, so Welch’s method averages across segments to estimate the average power at each frequency. For more about Welch’s method, see <http://thinkcomplex.com/welch>.

The parameter `nperseg` specifies the number of time steps per segment. With longer segments, we can estimate the power for more frequencies. With shorter segments, we get better estimates for each frequency. The value I chose, 2048, balances these tradeoffs.

The parameter `fs` is the “sampling frequency”, which is the number of data points in the signal per unit of time. By setting `fs=nperseg`, we get a range of frequencies from 0 to `nperseg/2`. This range is convenient, but because the units of time in the model are arbitrary, it doesn’t mean much.

The return values, `freqs` and `powers`, are NumPy arrays containing the frequencies of the components and their corresponding powers, which we can plot. Figure 8.6 shows the result.

For frequencies between 10 and 1000 (in arbitrary units), the spectrum falls on a straight line, which is what we expect for pink or red noise.

The gray line in the figure has slope  $-1.58$ , which indicates that

$$\log P(f) \sim -\beta \log f$$

with parameter  $\beta = 1.58$ , which is the same parameter reported by Bak, Tang, and Wiesenfeld. This result confirms that the sand pile model generates pink noise.

```
2.opted seq
```

要计算这个信号的功率谱，我们可以使用 SciPy 函数 `welch`:

```
来自 scipy 信号输入 welch
```

```
2048
```

```
Freqs, spectrum = welch(signal, nperseg = nperseg, fs = nperseg)
```

该函数采用韦尔奇的方法，将信号分成若干段，并计算每个段的功率谱。结果通常是嘈杂的，所以韦尔奇的方法平均跨部门估计每个频率的平均功率。想了解更多 Welch 的方法，请参阅 <http://thinkcomplex>。我是韦尔奇。

参数 `nperseg` 专门表示每个段的时间步数。使用较长的段，我们可以估计更多频率的功率。对于较短的段，我们可以更好地估计每个频率。我选择的价值，2048，平衡了这些交易。

参数 `fs` 是采样频率”，即每单位时间内信号中的数据点数。通过设置 `fs = nperseg`，我们得到了从 0 到 `nperseg/2` 的频率范围。这个范围很方便，但是因为模型中的时间单位是任意的，所以它没有多大意义。

返回值 `freqs` 和 `spectrum` 是 NumPy 数组，包含组件的频率及其对应的幂，我们可以绘制这些值。图 8.6 显示了结果。

对于频率在 10 到 1000 之间(任意单位)，频谱落在一条直线上，这就是我们所期望的粉红色或红色噪声。

图中灰线的斜率为 1:58，表明对数  $p(f) \log f$

参数 = 1:58，这与 Bak, Tang 和 Wiesenfeld 报告的参数相同。结果表明，砂桩模型产生粉红噪声。

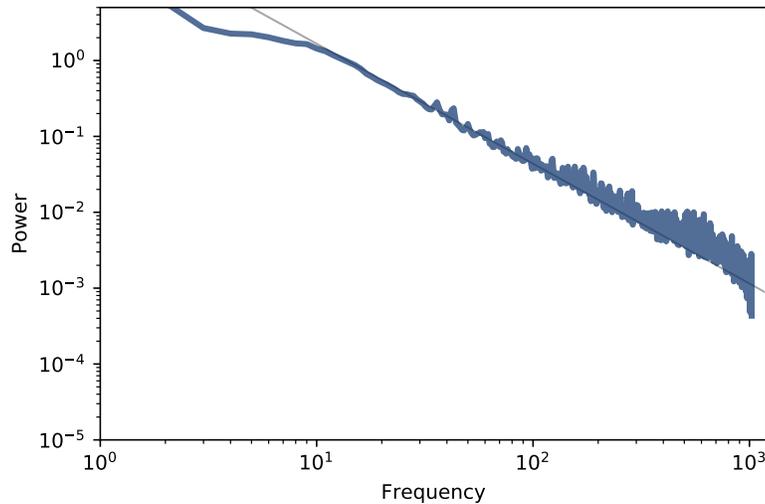


Figure 8.6: Power spectrum of the number of toppled cells over time, log-log scale.

## 8.8 Reductionism and Holism

The original paper by Bak, Tang and Wiesenfeld is one of the most frequently-cited papers in the last few decades. Some subsequent papers have reported other systems that are apparently self-organized critical (SOC). Others have studied the sand pile model in more detail.

As it turns out, the sand pile model is not a good model of a sand pile. Sand is dense and not very sticky, so momentum has a non-negligible effect on the behavior of avalanches. As a result, there are fewer very large and very small avalanches than the model predicts, and the distribution might not be heavy-tailed.

Bak has suggested that this observation misses the point. The sand pile model is not meant to be a realistic model of a sand pile; it is meant to be a simple example of a broad category of models.

To understand this point, it is useful to think about two kinds of models, **reductionist** and **holistic**. A reductionist model describes a system by describing its parts and their interactions. When a reductionist model is used

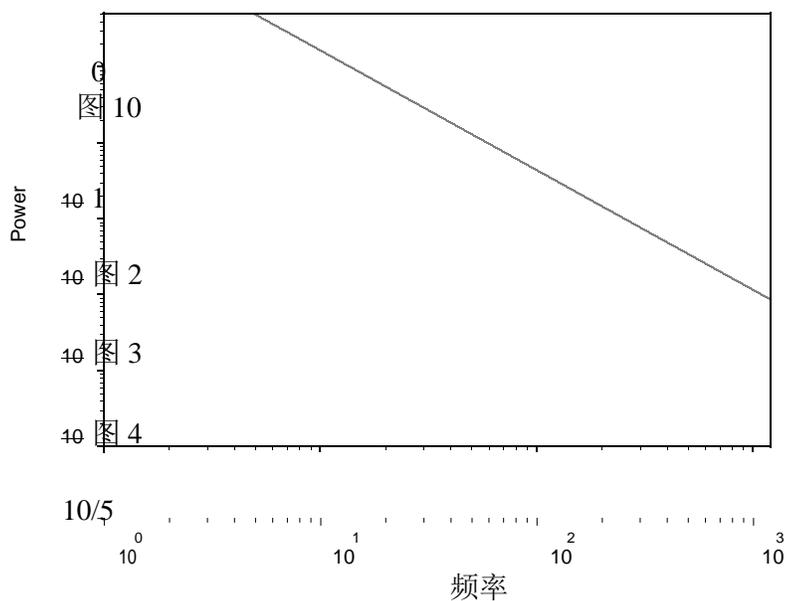


图 8.6: 倒塌细胞数量随时间变化的功率谱，对数刻度。

## 8.8 还原论与整体论

贝克、唐和维森菲尔德的原创论文是过去几十年中被引用最多的论文之一。随后的一些论文报道了其他明显是自组织批判(SOC)的系统。其他学者对砂堆模型进行了较为详细的研究。

事实证明，沙堆模型并不是一个好的沙堆模型。沙子密度大，粘性不大，因此动量对雪崩的行为有着不可忽视的影响。因此，极大极小雪崩的数量比模型预测的要少，而且分布可能不是重尾分布。

贝克认为这种观察没有抓住要点。沙堆模型并不是一个真实的沙堆模型，而是一个简单的模型范例。

为了理解这一点，我们需要思考两种模型，简化论和整体论。简化模型通过描述系统的各个部分及其相互作用来描述系统。当使用简化模型时

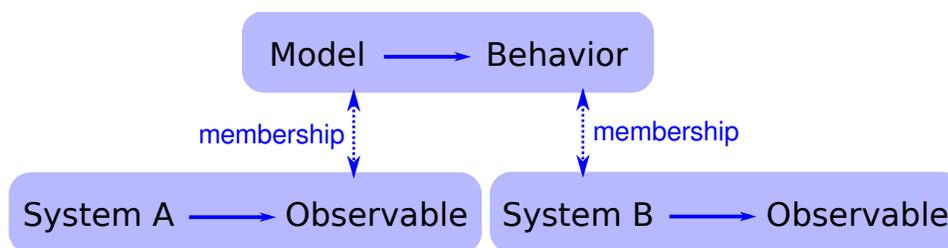


Figure 8.7: The logical structure of a holistic model.

as an explanation, it depends on an analogy between the components of the model and the components of the system.

For example, to explain why the ideal gas law holds, we can model the molecules that make up a gas with point masses and model their interactions as elastic collisions. If you simulate or analyze this model, you find that it obeys the ideal gas law. This model is satisfactory to the degree that molecules in a gas behave like molecules in the model. The analogy is between the parts of the system and the parts of the model.

Holistic models are more focused on similarities between systems and less interested in analogous parts. A holistic approach to modeling consists of these steps:

- Observe a behavior that appears in a variety of systems.
- Find a simple model that demonstrates that behavior.
- Identify the elements of the model that are necessary and sufficient to produce the behavior.

For example, in *The Selfish Gene*, Richard Dawkins suggests that genetic evolution is just one example of an evolutionary system. He identifies the essential elements of the category — discrete replicators, variability, and differential reproduction — and proposes that any system with these elements will show evidence of evolution.

As another example of an evolutionary system, he proposes “memes”, which are thoughts or behaviors that are replicated by transmission from person to

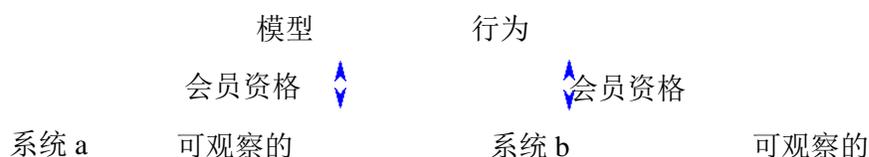


图 8.7: 整体模型的逻辑结构。

作为一种解释，它依赖于模型组件和系统组件之间的类比。

例如，为了解释为什么理想气体定律成立，我们可以用点质量来模拟组成气体的分子，并用弹性碰撞来模拟它们之间的相互作用。如果你模拟或分析这个模型，你会发现它遵循理想气体定律。这个模型对于气体中的分子在模型中表现得像分子一样的程度是令人满意的。这个类比是在系统的各个部分和模型的各个部分之间。

整体模型更注重系统之间的相似性，而对类似部分的兴趣较少。建模的整体方法包括以下步骤：

观察出现在各种系统中的行为。

找到一个简单的模型来演示这种行为。

确定模型中必要的元素以及产生行为的支持元素。

例如，在《塞尔什基因》一书中，理查德·道金斯认为基因进化只是进化系统的一个例子。他指出了分类的基本要素 | 离散的复制因子，可变性和双向繁殖 | 并提出任何具有这些要素的系统都将显示进化的证据。

作为进化系统的另一个例子，他提出了模因，即通过人与人之间的传播复制的思想或行为

person<sup>2</sup>. As memes compete for the resource of human attention, they evolve in ways that are similar to genetic evolution.

Critics of the meme model have pointed out that memes are a poor analogy for genes; they differ from genes in many obvious ways. Dawkins has argued that these differences are beside the point because memes are not *supposed* to be analogous to genes. Rather, memes and genes are examples of the same category: evolutionary systems. The differences between them emphasize the real point, which is that evolution is a general model that applies to many seemingly disparate systems. The logical structure of this argument is shown in Figure 8.7.

Bak has made a similar argument that self-organized criticality is a general model for a broad category of systems:

Since these phenomena appear everywhere, they cannot depend on any specific detail whatsoever... If the physics of a large class of problems is the same, this gives [the theorist] the option of selecting the *simplest* possible [model] belonging to that class for detailed study.<sup>3</sup>

Many natural systems demonstrate behaviors characteristic of critical systems. Bak's explanation for this prevalence is that these systems are examples of the broad category of self-organized criticality. There are two ways to support this argument. One is to build a realistic model of a particular system and show that the model exhibits SOC. The second is to show that SOC is a feature of many diverse models, and to identify the essential characteristics those models have in common.

The first approach, which I characterize as reductionist, can explain the behavior of a particular system. The second approach, which I am calling holistic, can explain the prevalence of criticality in natural systems. They are different models with different purposes.

For reductionist models, realism is the primary virtue, and simplicity is secondary. For holistic models, it is the other way around.

---

<sup>2</sup>This use of "meme" is original to Dawkins, and predates the distantly-related use of the word on the Internet by about 20 years.

<sup>3</sup>Bak, *How Nature Works*, Springer-Verlag 1996, page 43.

2号人物。当模因争夺人类注意力资源时，它们的进化方式类似于基因进化。

模因模型的批评者指出，模因与基因的类比很差；它们在许多明显的方面与基因脱节。道金斯认为这些差异并不重要，因为模因并不应该与基因类似。相反，模因和基因是同一类别的例子：进化系统。它们之间的差异强调了真正的意义，即进化是一个普遍的模式，适用于许多看似不同的系统。这个参数的逻辑结构如图 8.7 所示。

贝克也提出了类似的观点，认为自组织临界性是一个广义系统的通用模型：

由于这些现象无处不在，它们不能依赖于任何特定的细节... .. 如果一大类问题的物理学是相同的，这给了[理论家]选择最简单的可能[模型]属于该类的详细研究。图 3

许多自然系统表现出临界系统的行为特征。贝克对这种流行的解释是，这些系统就是自组织临界性的广义范畴的例子。有两种方法可以支持这种观点。一种是建立一个特定系统的真实模型，并表明该模型具有 SOC 特性。第二是说明 SOC 是许多不同模型的一个特征，并识别这些模型共有的本质特征。

第一种方法，我把它描述为还原论者，可以解释特定系统的行为。第二种方法，我称之为整体论，可以解释自然系统中临界性的盛行。它们是有着不同目的的不同模型。

对于简化模型来说，现实主义是主要的优点，简单性是次要的。对于整体模型来说，情况恰恰相反。

---

<sup>2</sup>“模因”的这种用法是道金斯最早使用的，比这个词在互联网上的使用还要早 20 年。

<sup>3</sup> Bak, *How Nature Works*, Springer-Verlag 1996, page 43.

## 8.9 SOC, causation, and prediction

If a stock market index drops by a fraction of a percent in a day, there is no need for an explanation. But if it drops 10%, people want to know why. Pundits on television are willing to offer explanations, but the real answer may be that there is no explanation.

Day-to-day variability in the stock market shows evidence of criticality: the distribution of value changes is heavy-tailed and the time series exhibits pink noise. If the stock market is a critical system, we should expect occasional large changes as part of the ordinary behavior of the market.

The distribution of earthquake sizes is also heavy-tailed, and there are simple models of the dynamics of geological faults that might explain this behavior. If these models are right, they imply that large earthquakes are not exceptional; that is, they do not require explanation any more than small earthquakes do.

Similarly, Charles Perrow has suggested that failures in large engineered systems, like nuclear power plants, are like avalanches in the sand pile model. Most failures are small, isolated, and harmless, but occasionally a coincidence of bad fortune yields a catastrophe. When big accidents occur, investigators go looking for the cause, but if Perrow's "normal accident theory" is correct, there may be no special cause of large failures.

These conclusions are not comforting. Among other things, they imply that large earthquakes and some kinds of accidents are fundamentally unpredictable. It is impossible to look at the state of a critical system and say whether a large avalanche is "due". If the system is in a critical state, then a large avalanche is always possible. It just depends on the next grain of sand.

In a sand pile model, what is the cause of a large avalanche? Philosophers sometimes distinguish the **proximate** cause, which is most immediately responsible, from the **ultimate** cause, which is considered some deeper kind of explanation (see <http://thinkcomplex.com/cause>).

In the sand pile model, the proximate cause of an avalanche is a grain of sand, but the grain that causes a large avalanche is identical to every other grain, so it offers no special explanation. The ultimate cause of a large avalanche is the structure and dynamics of the systems as a whole: large avalanches occur because they are a property of the system.

### 8.9 SOC, 因果关系和预测

如果一个股票市场指数在一天内下跌了百分之零点几，那就没有必要解释了。但是如果它下降了 10%，人们想知道为什么。电视上的专家们愿意做出解释，但真正的答案可能是没有解释。

股票市场的日常变化显示了临界性的证据：价值变化的分布是重尾分布，时间序列显示出粉红噪音。如果股票市场是一个关键的系统，我们应该期待偶尔的大的变化作为市场的普通行为的一部分。

地震规模的分布也是重尾分布的，有一些简单的地质断层动力学模型可以解释这种行为。如果这些模型是正确的，那么它们意味着大地震并非例外；也就是说，它们并不比小地震更需要解释。

类似地，查尔斯·珀罗认为，大型工程系统的故障，如核电站，就像沙堆模型中的雪崩。大多数的失败都是小的、孤立的、无害的，但是偶尔坏运气的巧合会导致一场灾难。当重大事故发生时，调查人员会去寻找原因，但如果珀罗的“正常事故理论”是正确的，可能就没有特殊的原因导致重大事故。

这些结论令人不安。在其他方面，他们暗示，大地震和某些类型的事故是根本不可预测的。观察一个关键系统的状态，判断是否会发生大规模雪崩，是不可能的。”。如果系统处于临界状态，那么大规模的雪崩总是可能的。这取决于下一粒沙子。

在沙堆模型中，大雪崩的原因是什么？哲学家有时会区分近因和终极原因，前者是最直接的原因，后者被认为是更深层次的解释(见 <http://thinkcomplex.com/cause>)。

在沙堆模型中，造成雪崩的直接原因是一粒沙子，但是引起大规模雪崩的颗粒与其他颗粒是相同的，所以没有特别的解释。大雪崩的最终原因是系统作为一个整体的结构和动力学：大雪崩的发生是因为它们是系统的一个属性。

Many social phenomena, including wars, revolutions, epidemics, inventions, and terrorist attacks, are characterized by heavy-tailed distributions. If these distributions are prevalent because social systems are SOC, major historical events may be fundamentally unpredictable and unexplainable.

## 8.10 Exercises

The code for this chapter is in the Jupyter notebook `chap08.ipynb` in the repository for this book. Open this notebook, read the code, and run the cells. You can use this notebook to work on the following exercises. My solutions are in `chap08soln.ipynb`.

**Exercise 8.1** To test whether the distributions of **T** and **S** are heavy-tailed, we plotted their PMFs on a log-log scale, which is what Bak, Tang and Wiesenfeld show in their paper. But as we saw in Section 4.7, this visualization can obscure the shape of the distribution. Using the same data, make a plot that shows the cumulative distributions (CDFs) of **S** and **T**. What can you say about their shape? Do they follow a power law? Are they heavy-tailed?

You might find it helpful to plot the CDFs on a log-x scale and on a log-log scale.

**Exercise 8.2** In Section 8.5 we showed that the initial configuration of the sand pile model produces fractal patterns. But after we drop a large number of random grains, the patterns look more random.

Starting with the example in Section 8.5, run the sand pile model for a while and then compute fractal dimensions for each of the 4 levels. Is the sand pile model fractal in steady state?

**Exercise 8.3** Another version of the sand pile model, called the “single source” model, starts from a different initial condition: instead of all cells at the same level, all cells are set to 0 except the center cell, which is set to a large value. Write a function that creates a `SandPile` object, sets up the single source initial condition, and runs until the pile reaches equilibrium. Does the result appear to be fractal?

You can read more about this version of the sand pile model at <http://thinkcomplex.com/sand>.

许多社会现象，包括战争、革命、流行病、发明和恐怖袭击，都是拥有属性分布。如果这些分布是普遍的，因为社会系统是 SOC，主要的历史事件可能是根本不可预测和无法解释的。

### 8.10 练习

本章的代码在本书资料库中的 `chapyter` 笔记本 `chap08.ipynb` 中。打开这个笔记本，阅读代码，并运行单元格。你可以用这个笔记本做以下练习。我的解决方案在第 8 章。

练习 8.1 为了检验  $t$  和  $s$  的分布是否为重尾分布，我们在对数尺度上绘制了它们的 PMFs, Bak, Tang 和 Wiesenfeld 在他们的论文中也显示了这一点。但是正如我们在第 4.7 节中看到的，这种可视化可以掩盖分布的形状。使用相同的数据，绘制一张图表，显示  $s$  和  $t$  的累积分布(CDFs)。你能说说它们的形状吗？他们遵循幂定律吗？它们是厚尾的吗？

您可能会发现，将 CDFs 绘制在  $\log-x$  尺度和  $\log\text{-}\log$  尺度上是有帮助的。

练习 8.2 在第 8.5 节中，我们展示了砂堆模型的初始饱和度产生了分形图案。但是当我们丢掉大量的随机颗粒后，这些模式看起来就更随机了。

从第 8.5 节的例子开始，运行沙堆模型一段时间，然后计算 4 个层次中每一层的分形维数。砂堆模型是稳态的分形吗？

练习 8.3 沙堆模型的另一个版本，称为单源模型，从一个不同的初始条件开始：除了中心单元格设置为大值之外，所有单元格都设置为 0，而不是处于同一水平的所有单元格。编写一个函数，创建一个 `SandPile` 对象，设置单源初始条件，并运行到桩达到平衡。结果是不是不规则的呢？

你可以阅读更多关于这个版本的沙堆模型在 [http:// thinkcomplex.com/sand](http://thinkcomplex.com/sand)。

**Exercise 8.4** In their 1989 paper, Bak, Chen and Creutz suggest that the Game of Life is a self-organized critical system (see <http://thinkcomplex.com/bak89>).

To replicate their tests, start with a random configuration and run the GoL CA until it stabilizes. Then choose a random cell and flip it. Run the CA until it stabilizes again, keeping track of  $T$ , the number of time steps it takes, and  $S$ , the number of cells affected. Repeat for a large number of trials and plot the distributions of  $T$  and  $S$ . Also, estimate the power spectrums of  $T$  and  $S$  as signals in time, and see if they are consistent with pink noise.

**Exercise 8.5** In *The Fractal Geometry of Nature*, Benoit Mandelbrot proposes what he calls a “heretical” explanation for the prevalence of heavy-tailed distributions in natural systems. It may not be, as Bak suggests, that many systems can generate this behavior in isolation. Instead there may be only a few, but interactions between systems might cause the behavior to propagate.

To support this argument, Mandelbrot points out:

- The distribution of observed data is often “the joint effect of a fixed underlying *true distribution* and a highly variable *filter*”.
- Heavy-tailed distributions are robust to filtering; that is, “a wide variety of filters leave their asymptotic behavior unchanged”.

What do you think of this argument? Would you characterize it as reductionist or holist?

**Exercise 8.6** Read about the “Great Man” theory of history at <http://thinkcomplex.com/great>. What implication does self-organized criticality have for this theory?

练习 8.4 Bak, Chen 和 Creutz 在他们 1989 年的论文中指出, 生命的游戏是一个自我组织的批判系统(参见 <http://thinkcomplex.com/bak89>).

为了复制他们的测试, 从随机调节开始, 运行 GoL CA 直到稳定下来。然后随机选择一个单元格并激活它。运行 CA 直到它再次稳定下来, 记录  $t$ , 它采取的时间步数, 和  $s$ , 被选择的细胞数。重复大量试验, 绘制  $t$  和  $s$  的分布图。同时, 在时间上估计  $t$  和  $s$  的功率谱, 看看它们是否与粉红噪声一致。

练习 8.5 在《自然的分形几何》一书中, 本华·曼德博提出了他称之为异端学说的解释, 来解释重尾分布在自然系统中的盛行。可能不像 Bak 所说的那样, 许多系统可以独立地生成这种行为。相反, 可能只有少数, 但系统之间的相互作用可能导致行为的传播。

为了支持这一观点, 曼德布洛特指出:

观测资料的分布往往是一个整体的结果  
潜在的真实分布和高度可变的  $l_{iter}$ ”。

重尾分布对散射是鲁棒的, 也就是说, 各种各样的散射保持它们的渐近行为不变。

你如何看待这个论点? 你认为它是简化论者还是整体论者?

练习 8.6 阅读关于伟人的历史理论 [http:// thinkcomplex.com/Great](http://thinkcomplex.com/Great)。自组织临界性对这个理论有什么启示?





# Chapter 9

## Agent-based models

The models we have seen so far might be characterized as “rule-based” in the sense that they involve systems governed by simple rules. In this and the following chapters, we explore **agent-based models**.

Agent-based models include **agents** that are intended to model people and other entities that gather information about the world, make decisions, and take actions.

The agents are usually situated in space or in a network, and interact with each other locally. They usually have imperfect or incomplete information about the world.

Often there are differences among agents, unlike previous models where all components are identical. And agent-based models often include randomness, either among the agents or in the world.

Since the 1970s, agent-based modeling has become an important tool in economics, other social sciences, and some natural sciences.

Agent-based models are useful for modeling the dynamics of systems that are not in equilibrium (although they are also used to study equilibrium). And they are particularly useful for understanding relationships between individual decisions and system behavior.

The code for this chapter is in `chap09.ipynb`, which is a Jupyter notebook in the repository for this book. For more information about working with this code, see Section 0.3.

## 第九章

### 基于 agent 的模型

到目前为止，我们看到的模型可能被描述为基于规则的”，因为它们涉及由简单规则管理的系统。在本章和后面的章节中，我们探讨了基于 agent 的模型。

基于代理的模型包括代理，这些代理旨在为收集世界信息、做出决策和采取行动的人和其他实体建模。

这些代理通常位于空间或网络中，并且在局部地相互作用。他们通常对世界有不完善或不完整的了解。

不像以前的模型，所有的成分都是相同的。而基于主体的模型通常包含随机性，无论是在主体之间还是在世界上。

自 20 世纪 70 年代以来，基于主体的建模已经成为经济学、其他社会科学和一些自然科学的重要工具。

基于 agent 的模型对于建立非平衡系统的动力学模型非常有用(尽管它们也用于研究平衡)。它们对于理解个人决策和系统行为之间的关系特别有用。

本章的代码在 chap09.ipynb 中，这是本书资料库中的一个 Jupyter 笔记本。有关使用此代码的更多信息，请参见 0.3 节。

## 9.1 Schelling's Model

In 1969 Thomas Schelling published “Models of Segregation”, which proposed a simple model of racial segregation. You can read it at <http://thinkcomplex.com/schell>.

The Schelling model of the world is a grid where each cell represents a house. The houses are occupied by two kinds of agents, labeled red and blue, in roughly equal numbers. About 10% of the houses are empty.

At any point in time, an agent might be happy or unhappy, depending on the other agents in the neighborhood, where the “neighborhood” of each house is the set of eight adjacent cells. In one version of the model, agents are happy if they have at least two neighbors like themselves, and unhappy if they have one or zero.

The simulation proceeds by choosing an agent at random and checking to see whether they are happy. If so, nothing happens; if not, the agent chooses one of the unoccupied cells at random and moves.

You will not be surprised to hear that this model leads to some segregation, but you might be surprised by the degree. From a random starting point, clusters of similar agents form almost immediately. The clusters grow and coalesce over time until there are a small number of large clusters and most agents live in homogeneous neighborhoods.

If you did not know the process and only saw the result, you might assume that the agents were racist, but in fact all of them would be perfectly happy in a mixed neighborhood. Since they prefer not to be greatly outnumbered, they might be considered mildly xenophobic. Of course, these agents are a wild simplification of real people, so it may not be appropriate to apply these descriptions at all.

Racism is a complex human problem; it is hard to imagine that such a simple model could shed light on it. But in fact it provides a strong argument about the relationship between a system and its parts: if you observe segregation in a real city, you cannot conclude that individual racism is the immediate cause, or even that the people in the city are racists.

### 9.1 谢林的模型

1969 年，托马斯·克洛姆比·谢林出版了《种族隔离模型》，提出了一个简单的种族隔离模型。你可以在 [http:// thinkcomplex.com/schell](http://thinkcomplex.com/schell) 上阅读。

谢林的世界模型是一个网格，每个单元代表一个房子。这些房子由两种代理人居住，红色和蓝色的代理人大致相等。大约 10% 的房子是空的。

在任何时候，一个代理人都可能是高兴或不高兴，这取决于邻居中的其他代理人，其中每个房子的邻居是八个相邻单元的集合。在这个模型的一个版本中，如果至少有两个像自己一样的邻居，代理人会感到高兴；如果只有一个或零个邻居，代理人会感到不高兴。

模拟通过随机选择一个代理并检查它们是否满意来进行。如果是这样，什么也不会发生；如果不是这样，代理随机选择一个空闲的单元格并移动。

听到这种模式导致了种族隔离，你不会感到惊讶，但是你可能会对它的程度感到惊讶。从一个随机的起点，相似的代理几乎立即形成簇。随着时间的推移，这些集群不断成长和合并，直到出现少数大型集群，而且大多数代理都生活在同质的社区中。

如果你不知道这个过程，只看到结果，你可能会认为这些特工是种族主义者，但事实上，他们所有人在一个混杂的社区里都会非常快乐。由于他们不希望在数量上被远远超过，他们可能被认为是轻度的排外。当然，这些代理人是真实人物的狂野简单化，所以根本不适合应用这些描述。

种族主义是一个复杂的人类问题；很难想象这样一个简单的模型能够阐明这个问题。但事实上，它提供了一个强有力的论点之间的关系系统及其部分：如果你观察隔离在一个真正的城市，你不能得出结论，个人种族主义是直接原因，或甚至在城市的人是种族主义者。

Of course, we have to keep in mind the limitations of this argument: Schelling's model demonstrates a possible cause of segregation, but says nothing about actual causes.

## 9.2 Implementation of Schelling's model

To implement Schelling's model, I wrote yet another class that inherits from `Cell2D`:

```
class Schelling(Cell2D):  
  
    def __init__(self, n, p):  
        self.p = p  
        choices = [0, 1, 2]  
        probs = [0.1, 0.45, 0.45]  
        self.array = np.random.choice(choices, (n, n), p=probs)
```

`n` is the size of the grid, and `p` is the threshold on the fraction of similar neighbors. For example, if `p=0.3`, an agent will be unhappy if fewer than 30% of their neighbors are the same color.

`array` is a NumPy array where each cell is 0 if empty, 1 if occupied by a red agent, and 2 if occupied by a blue agent. Initially 10% of the cells are empty, 45% red, and 45% blue.

The `step` function for Schelling's model is substantially more complicated than previous examples. If you are not interested in the details, you can skip to the next section. But if you stick around, you might pick up some NumPy tips.

First, I make boolean arrays that indicate which cells are red, blue, and empty:

```
a = self.array  
red = a==1  
blue = a==2  
empty = a==0
```

Then I use `correlate2d` to count, for each location, the number of neighboring cells that are red, blue, and non-empty. We saw `correlate2d` in Section 6.6.

当然，我们必须牢记这个论点的局限性：谢林的模型证明了种族隔离的一个可能的原因，但没有说明实际的原因。

## 9.2 谢林模型的实现

为了实现谢林的模型，我编写了另一个继承自 `Cell2D`：

```
谢林班(Cell2D)：
```

```
返回文章页面【一分钟科普】：
```

```
    P = p
```

```
    选择 = [0,1,2]
```

```
    Probs = [0.1,0.45,0.45]
```

```
    Self.array = np.random.choice(choices, (n, n), p = probs)
```

`N` 是网格的大小，`p` 是相似邻居的门槛。例如，如果 `p = 0.3`，一个代理将不高兴，如果少于 30% 的邻居是相同的颜色。

`Array` 是 NumPy 数组，其中每个单元格为 0(如果为空)，1(如果为红色代理)，2(如果为蓝色代理)。最初 10% 的电池是空的，45% 是红色的，45% 是蓝色的。

谢林模型中的步长函数要比以前的例子复杂得多。如果您对细节不感兴趣，可以跳到下一节。但是如果你留下来，你可能会学到一些 NumPy 技巧。

首先，我创建一个布尔数组来指示哪些单元格是红色、蓝色和空的：

```
    A = self.array
```

```
    红 = a = 1
```

```
    蓝色 = a = 2
```

```
    空 = a = 0
```

然后我使用 `correlate2d` 来计算，对于每个位置，相邻的红色、蓝色和非空的单元格的数量。我们在 6.6 节中看到了相关性 `2d`。

```

options = dict(mode='same', boundary='wrap')

kernel = np.array([[1, 1, 1],
                  [1, 0, 1],
                  [1, 1, 1]], dtype=np.int8)

num_red = correlate2d(red, kernel, **options)
num_blue = correlate2d(blue, kernel, **options)
num_neighbors = num_red + num_blue

```

`options` is a dictionary that contains the options we pass to `correlate2d`. With `mode='same'`, the result is the same size as the input. With `boundary='wrap'`, the top edge is wrapped to meet the bottom, and the left edge is wrapped to meet the right.

`kernel` indicates that we want to consider the eight neighbors that surround each cell.

After computing `num_red` and `num_blue`, we can compute the fraction of neighbors, for each location, that are red and blue.

```

frac_red = num_red / num_neighbors
frac_blue = num_blue / num_neighbors

```

Then, we can compute the fraction of neighbors, for each agent, that are the same color as the agent. I use `np.where`, which is like an element-wise `if` expression. The first parameter is a condition that selects elements from the second or third parameter.

```

frac_same = np.where(red, frac_red, frac_blue)
frac_same[empty] = np.nan

```

In this case, wherever `red` is `True`, `frac_same` gets the corresponding element of `frac_red`. Where `red` is `False`, `frac_same` gets the corresponding element of `frac_blue`. Finally, where `empty` indicates that a cell is empty, `frac_same` is set to `np.nan`, which is a special value that indicates “Not a Number”.

Now we can identify the locations of the unhappy agents:

```

unhappy = frac_same < self.p
unhappy_locs = locs_where(unhappy)

```

```
选项 = dict (mode = " same" , boundary = " wrap")

Kernel = np.array ([1,1,1] ,
                   [1, 0, 1],
                   [1, 1, 1]), dtype=np.int8)
```

```
Num red = correlate2d (red, kernel, ** options) num blue =
correlate2d (blue, kernel, ** options) num neighbors = num
red + num blue
```

选项是一个包含我们传递给 `correlate2d` 的选项的字典。对于 `mode = " same"`，结果与输入的大小相同。使用 `boundary = ' wrap'`，顶部边缘被包装以满足底部，左边缘被包装以满足右部。

内核表明我们想要考虑每个单元周围的八个邻居。

在计算 `numred` 和 `numblue` 之后，我们可以计算每个位置的邻居的分数，它们是红色和蓝色的。

```
红色 = num red/num neighbors
蓝色，蓝色，蓝色，邻居
```

然后，我们可以计算邻居的分数，为每个代理，是相同的颜色，作为代理。我使用 `np.where`，它类似于元素级的 `if` 表达式。`Rst` 参数是从第二个或第三个参数中选择元素的条件。

```
Frac _ same = np.where (red, frac _ red, frac _ blue)
Frac _ same [ empty ] = np.nan
```

在这种情况下，只要 `red` 是 `True`，`frac _ same` 就会得到对应的 `frac _ red` 元素。当红色为 `False` 时，`frac _ same` 得到相应的 `frac _ blue` 元素。最后，当 `empty` 表示单元格为空时，`frac _ same` 被设置为 `np.nan`，这是一个特殊值，表示 "Not a Number"。

现在我们可以确定不满意的代理人的位置:

```
不高兴 = frac 同样的 < self.p
不开心的疯子就是不开心的疯子
```

`locs_where` is a wrapper function for `np.nonzero`:

```
def locs_where(condition):  
    return list(zip(*np.nonzero(condition)))
```

`np.nonzero` takes an array and returns the coordinates of all non-zero cells; the result is a tuple of arrays, one for each dimension. Then `locs_where` uses `list` and `zip` to convert this result to a list of coordinate pairs.

Similarly, `empty_locs` is an array that contains the coordinates of the empty cells:

```
empty_locs = locs_where(empty)
```

Now we get to the core of the simulation. We loop through the unhappy agents and move them:

```
num_empty = np.sum(empty)  
for source in unhappy_locs:  
    i = np.random.randint(num_empty)  
    dest = empty_locs[i]  
  
    a[dest] = a[source]  
    a[source] = 0  
    empty_locs[i] = source
```

`i` is the index of a random empty cell; `dest` is a tuple containing the coordinates of the empty cell.

In order to move an agent, we copy its value (1 or 2) from `source` to `dest`, and then set the value of `source` to 0 (since it is now empty).

Finally, we replace the entry in `empty_locs` with `source`, so the cell that just became empty can be chosen by the next agent.

## 9.3 Segregation

Now let's see what happens when we run the model. I'll start with `n=100` and `p=0.3`, and run for 10 steps.

哪里是 `np.nonzero` 的包装函数:

条件:

```
Return list (zip (* np.nonzero (condition)))
```

`Nonzero` 接受一个数组并返回所有非零单元格的坐标; 结果是一个数组的元组, 每个维一个。然后 `locs_where` 使用 `list` 和 `zip` 将此结果转换为一个坐标对列表。

类似地, `empty_locs` 是一个包含空单元格坐标的数组:

```
Empty_locs = locs_where (empty)
```

现在我们进入模拟的核心, 我们循环遍历这些不开心的代理并移动它们:

```
Num_empty = np.sum (empty)
来源: 《闷闷不乐的疯子》
I = np.random.randint (num_empty)
[i] = empty_locs [i]
```

一个[最][来源]

```
A [来源] = 0
```

```
Empty_locs [i] = source
```

`I` 是随机空单元格的索引; `dest` 是包含空单元格坐标的元组。

为了移动代理, 我们将其值(1 或 2)从源文件复制到 `dest` 文件, 然后将源文件的值设置为 0(因为它现在是空的)。

最后, 我们将 `empty_locs` 中的条目替换为 `source`, 这样下一个代理就可以选择刚刚变为空的单元格。

### 9.3 种族隔离

现在让我们看看运行模型时会发生什么。我从  $n = 100$  和  $p = 0.3$  开始, 跑 10 步。

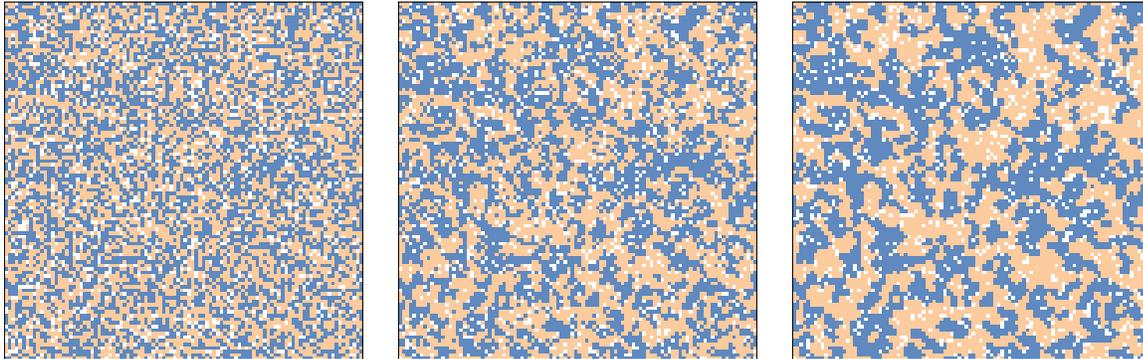


Figure 9.1: Schelling's segregation model with  $n=100$ , initial condition (left), after 2 steps (middle), and after 10 steps (right).

```
grid = Schelling(n=100, p=0.3)
for i in range(10):
    grid.step()
```

Figure 9.1 shows the initial configuration (left), the state of the simulation after 2 steps (middle), and the state after 10 steps (right).

Clusters form almost immediately and grow quickly, until most agents live in highly-segregated neighborhoods.

As the simulation runs, we can compute the degree of segregation, which is the average, across agents, of the fraction of neighbors who are the same color as the agent:

```
np.nanmean(frac_same)
```

In Figure 9.1, the average fraction of similar neighbors is 50% in the initial configuration, 65% after two steps, and 76% after 10 steps!

Remember that when  $p=0.3$  the agents would be happy if 3 of 8 neighbors were their own color, but they end up living in neighborhoods where 6 or 7 of their neighbors are their own color, typically.

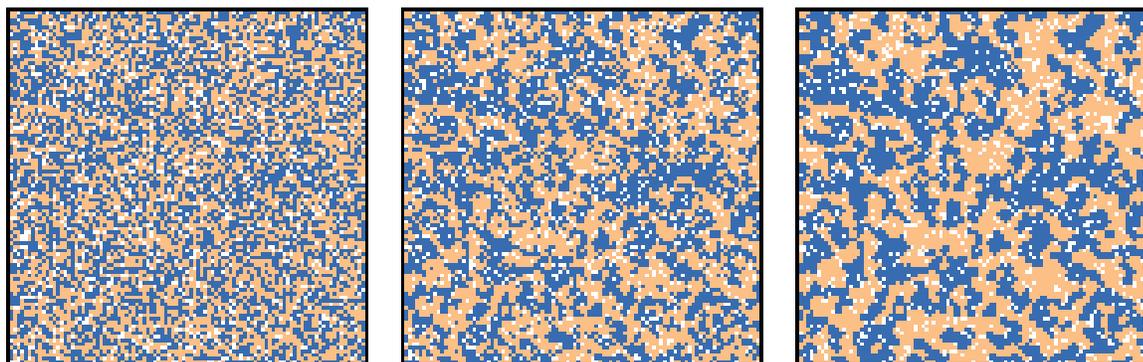


图 9.1: 谢林的分隔模型,  $n = 100$ , 初始条件(左), 经过两个步骤(中), 经过 10 个步骤(右)。

```
网格 = 谢林(n = 100, p = 0.3)
```

```
对于 i 在范围(10):
```

```
    Grid.step ()
```

图 9.1 显示了初始饱和度(左)、两个步骤后的模拟状态(中)和 10 个步骤后的状态(右)。

集群几乎立即形成, 并迅速增长, 直到大多数代理人居住在高度隔离的社区。

随着模拟程序的运行, 我们可以计算隔离的程度, 也就是隔离代理之间相同颜色的邻居比例的平均值:

```
Np.nanmean (frac _ same)
```

在图 9.1 中, 相似邻居的平均分数在初始饱和度中为 50% , 两步后为 65% , 10 步后为 76% !

记住, 当  $p = 0.3$  时, 如果 8 个邻居中有 3 个是他们自己的颜色, 经纪人会很高兴, 但是他们最终居住的社区有 6 或 7 个邻居是他们自己的颜色, 通常。

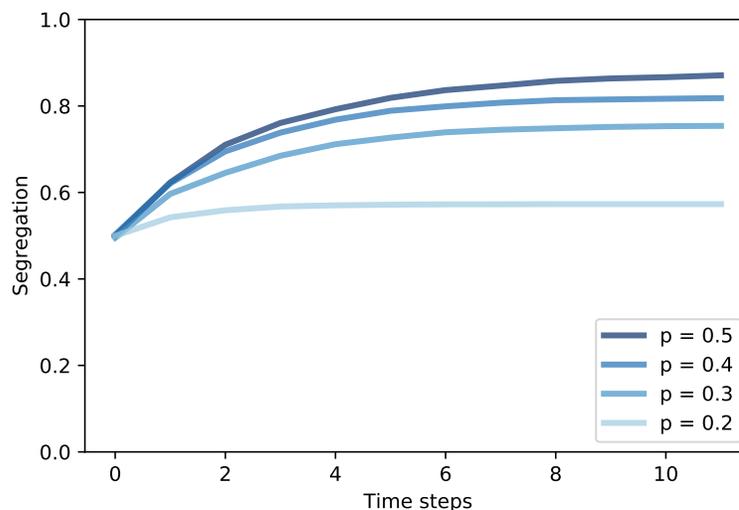


Figure 9.2: Degree of segregation in Schelling’s model, over time, for a range of  $p$ .

Figure 9.2 shows how the degree of segregation increases and where it levels off for several values of  $p$ . When  $p=0.4$ , the degree of segregation in steady state is about 82%, and a majority of agents have no neighbors with a different color.

These results are surprising to many people, and they make a striking example of the unpredictable relationship between individual decisions and system behavior.

## 9.4 Sugarscape

In 1996 Joshua Epstein and Robert Axtell proposed Sugarscape, an agent-based model of an “artificial society” intended to support experiments related to economics and other social sciences.

Sugarscape is a versatile model that has been adapted for a wide variety of topics. As examples, I will replicate the first few experiments from Epstein and Axtell’s book, *Growing Artificial Societies*.

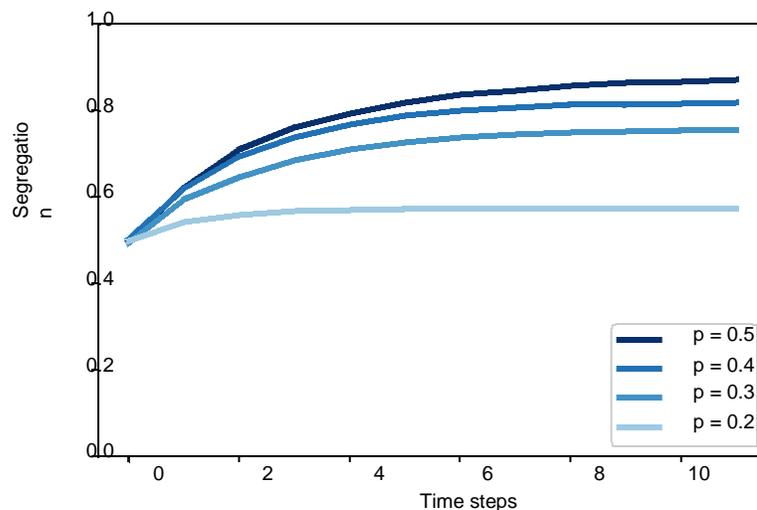


图 9.2: 在谢林的模型中，随着时间的推移，在  $p$  的范围内的隔离程度。

图 9.2 显示了隔离的程度如何增加，以及  $p$  的几个值在哪里达到  $\alpha$  级。当  $p = 0.4$  时，稳态偏析程度约为 82%，绝大多数试剂没有相邻颜色。

这些结果让许多人感到惊讶，它们为个人决策和系统行为之间不可预测的关系提供了一个惊人的例子。

#### 9.4 糖景

1996 年 Joshua Epstein 和 Robert Axtell 提出了 Sugarscape，一个社会学会的个体为本模型，旨在支持与经济学和其他社会科学相关的实验。

糖景是一个多用途的模式，已被改编为各种各样的主题。作为例子，我将重复爱泼斯坦和阿克斯特尔的书《成长的社会化社会》中的一些实验。

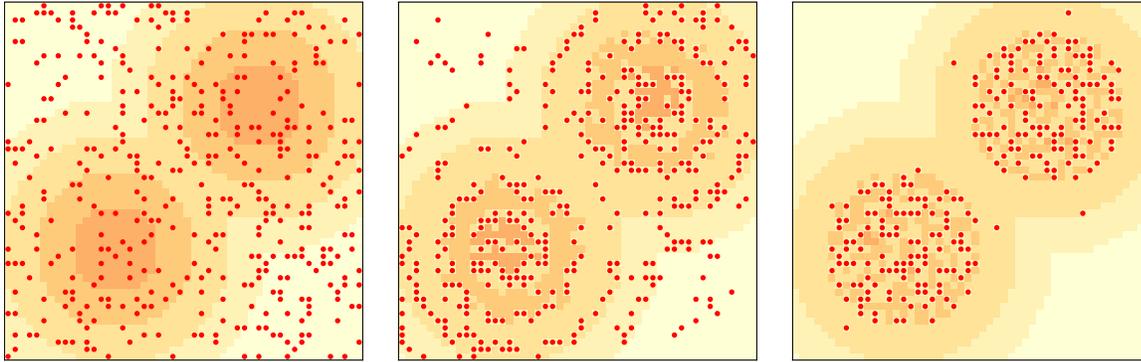


Figure 9.3: Replication of the original Sugarscape model: initial configuration (left), after 2 steps (middle) and after 100 steps (right).

In its simplest form, Sugarscape is a model of a simple economy where agents move around on a 2-D grid, harvesting and accumulating “sugar”, which represents economic wealth. Some parts of the grid produce more sugar than others, and some agents are better at finding it than others.

This version of Sugarscape is often used to explore and explain the distribution of wealth, in particular the tendency toward inequality.

In the Sugarscape grid, each cell has a capacity, which is the maximum amount of sugar it can hold. In the original configuration, there are two high-sugar regions, with capacity 4, surrounded by concentric rings with capacities 3, 2, and 1.

Figure 9.3 (left) shows the initial configuration, with the darker areas indicating cells with higher capacity, and small dots representing the agents.

Initially there are 400 agents placed at random locations. Each agent has three randomly-chosen attributes:

**Sugar:** Each agent starts with an endowment of sugar chosen from a uniform distribution between 5 and 25 units.

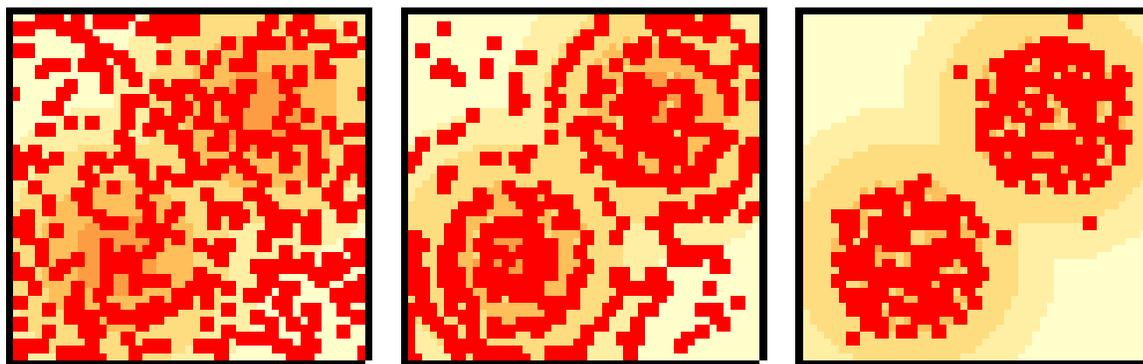


图 9.3: 原始 Sugarscape 模型的复制: 初始饱和(左)、两步(中)和 100 步(右)之后。

在最简单的形式中，“糖景”是一个简单经济模型，代理人在二维网格上移动，收获和积累糖”，这代表着经济财富。网格的某些部分比其他部分产生更多的糖，而且有些代理人比其他人更善于理解它。

这个版本的《甜景》经常被用来探索和解释财富的分配，特别是不平等的趋势。

在“糖景”网格中，每个单元格都有一个容量，这是它能容纳的最大糖量。在原始饱和度中，有两个高糖区，容量为 4，被容量为 3、2 和 1 的同心环包围。

图 9.3(左)显示了最初的饱和度，较暗的区域表示容量较大的细胞，小点表示代理。

最初有 400 个代理放置在随机位置。每个代理有三个随机选择的属性：

糖: 每个代理人从 5 到 25 个单位之间的均匀分布中选择一个糖储备。

**Metabolism:** Each agent has some amount of sugar they must consume per time step, chosen uniformly between 1 and 4.

**Vision:** Each agent can “see” the amount of sugar in nearby cells and move to the cell with the most, but some agents can see and move farther than others. The distance agents see is chosen uniformly between 1 and 6.

During each time step, agents move one at a time in a random order. Each agent follows these rules:

- The agent surveys  $k$  cells in each of the 4 compass directions, where  $k$  is the range of the agent’s vision.
- It chooses the unoccupied cell with the most sugar. In case of a tie, it chooses the closer cell; among cells at the same distance, it chooses randomly.
- The agent moves to the selected cell and harvests the sugar, adding the harvest to its accumulated wealth and leaving the cell empty.
- The agent consumes some part of its wealth, depending on its metabolism. If the resulting total is negative, the agent “starves” and is removed.

After all agents have executed these steps, the cells grow back some sugar, typically 1 unit, but the total sugar in each cell is bounded by its capacity.

Figure 9.3 (middle) shows the state of the model after two steps. Most agents are moving toward the areas with the most sugar. Agents with high vision move the fastest; agents with low vision tend to get stuck on the plateaus, wandering randomly until they get close enough to see the next level.

Agents born in the areas with the least sugar are likely to starve unless they have a high initial endowment and high vision.

Within the high-sugar areas, agents compete with each other to find and harvest sugar as it grows back. Agents with high metabolism or low vision are the most likely to starve.

When sugar grows back at 1 unit per time step, there is not enough sugar to sustain the 400 agents we started with. The population drops quickly at first, then more slowly, and levels off around 250.

新陈代谢: 每个代理有一定数量的糖, 他们必须消耗每一个时间步骤, 均匀选择 1 至 4。

视觉: 每个代理人可以看到附近细胞的糖的数量, 并移动到细胞与最, 但一些代理人可以看到和移动比其他人更远。所看到的距离代理均匀地选择在 1 和 6 之间。

在每个时间步骤中, 代理以随机顺序一次移动一个, 每个代理遵循以下规则:

该代理程序在 4 个指南针方向中的每一个方向上调查  $k$  细胞, 其中  $k$  是探员的视野范围。

它选择了含糖量最高的空置电池。在出现领带的情况下, 它选择距离较近的单元格; 在相同距离的单元格中, 它随机选择。

代理移动到选定的细胞和收获糖, 增加获得的财富, 其积累的财富, 离开细胞空。

代理人消耗它的部分财富, 这取决于它的新陈代谢。如果结果总数是负数, 代理饥饿”, 并被删除。

在所有代理程序完成这些步骤之后, 细胞会重新生成一些糖, 通常是一个单位, 但是每个细胞的总糖量受其容量的限制。

图 9.3(中间部分)显示了经过两个步骤后的模型状态。大多数代理商正在向含糖量最高的地区移动。具有高视力的代理人移动得最快; 具有低视力的代理人倾向于卡在高原上, 随机徘徊, 直到他们接近到足以看到下一个级别。

出生在糖分含量最低地区的代理人很可能会挨饿, 除非他们具有较高的初始禀赋和较高的视力。

在高糖地区, 代理商相互竞争, 以获得和收获糖, 因为它生长回来。新陈代谢高或视力低的病原体最有可能挨饿。

当糖以每个时间步长一个单位的速度增长时, 就没有足够的糖来维持我们开始使用的 400 个代理人。人口数量一开始下降得很快, 然后下降得更慢, 大约在 250 人左右。

Figure 9.3 (right) shows the state of the model after 100 time steps, with about 250 agents. The agents who survive tend to be the lucky ones, born with high vision and/or low metabolism. Having survived to this point, they are likely to survive forever, accumulating unbounded stockpiles of sugar.

## 9.5 Wealth inequality

In its current form, Sugarscape models a simple ecology, and could be used to explore the relationship between the parameters of the model, like the growth rate and the attributes of the agents, and the carrying capacity of the system (the number of agents that survive in steady state). And it models a form of natural selection, where agents with higher “fitness” are more likely to survive.

The model also demonstrates a kind of wealth inequality, with some agents accumulating sugar faster than others. But it would be hard to say anything specific about the distribution of wealth because it is not “stationary”; that is, the distribution changes over time and does not reach a steady state.

However, if we give the agents finite lifespans, the model produces a stationary distribution of wealth. Then we can run experiments to see what effect the parameters and rules have on this distribution.

In this version of the model, agents have an age that gets incremented each time step, and a random lifespan chosen from a uniform distribution between 60 to 100. If an agent’s age exceeds its lifespan, it dies.

When an agent dies, from starvation or old age, it is replaced by a new agent with random attributes, so the number of agents is constant.

Starting with 250 agents (which is close to carrying capacity) I run the model for 500 steps. After each 100 steps, I plot the cumulative distribution function (CDF) of sugar accumulated by the agents. We saw CDFs in Section 4.7. Figure 9.4 shows the results on a linear scale (left) and a log-x scale (right).

After about 200 steps (which is twice the longest lifespan) the distribution doesn’t change much. And it is skewed to the right.

Most agents have little accumulated wealth: the 25th percentile is about 10 and the median is about 20. But a few agents have accumulated much more: the 75th percentile is about 40, and the highest value is more than 150.

图 9.3(右)显示了经过 100 个时间步骤后的模型状态，其中包含大约 250 个代理。幸存者往往是幸运的，他们生来就有高视力和/或低新陈代谢。生存到这一点，他们可能会永远生存下去，积累无限的糖库存。

### 9.5 财富不平等

在现有的模型中，Sugarscape 模型是一个简单的生态学模型，可以用来研究模型参数(如生长速度和代理人的属性)与系统承载能力(在稳态下生存的代理人数量)之间的关系。它模拟了一种自然选择的形式，在这种形式下，具有更高特性的个体更有可能存活下来。

该模型还表明了一种财富不平等，一些代理人积累糖比其他人更快。但是财富的分配很难说有什么特别的，因为它不是稳定的”，也就是说，财富的分配会随着时间的推移而改变，不会达到稳定的状态。

然而，如果我们给代理人 nite 寿命，该模型产生一个稳定的财富分布。然后我们可以进行实验，看看参数和规则对这个分布有什么影响。

在这个版本的模型中，代理人的年龄每增加一个时间步骤，并且从 60 到 100 之间的均匀分布中选择一个随机的寿命。如果一个代理的寿命超过了它的寿命，它就会死亡。

当一个代理人死于饥饿或衰老时，它会被一个具有随机属性的新代理人取代，因此代理人的数量是恒定的。

从 250 个代理开始(接近承载能力)，我运行模型 500 步。在每 100 步之后，我绘制代理人所积累的糖的累积分布函数。我们在第 4.7 节中看到了 CDFs。图 9.4 显示了线性尺度(左)和 log-x 尺度(右)的结果。

经过大约 200 个步骤(这是最长寿命的两倍)后，分布没有太大变化。而且它是向右倾斜的。

大多数经纪人的积累财富很少: 第 25 个百分位数约为 10，中位数约为 20。但是一些代理商已经积累了更多的数据: 第 75 百分位数大约是 40，最高值超过 150。

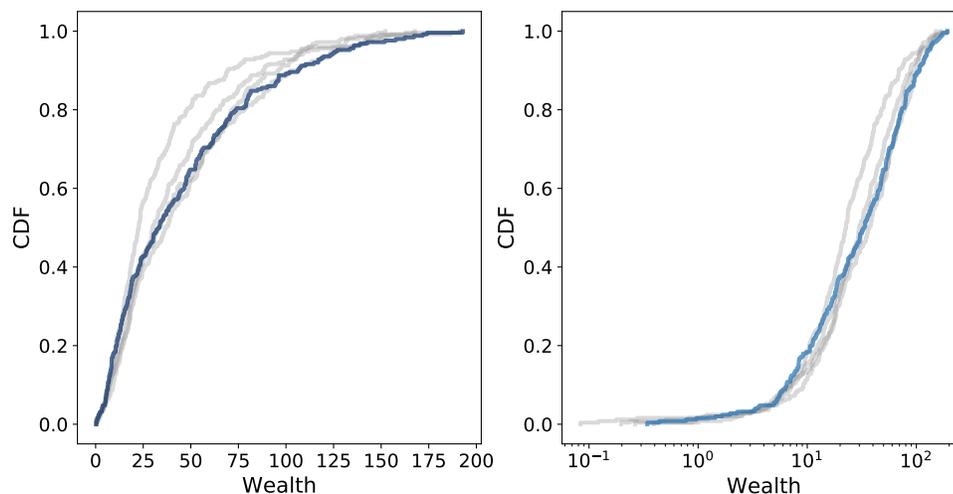


Figure 9.4: Distribution of sugar (wealth) after 100, 200, 300, and 400 steps (gray lines) and 500 steps (dark line). Linear scale (left) and log-x scale (right).

On a log scale the shape of the distribution resembles a Gaussian or normal distribution, although the right tail is truncated. If it were actually normal on a log scale, the distribution would be lognormal, which is a heavy-tailed distribution. And in fact, the distribution of wealth in practically every country, and in the world, is a heavy-tailed distribution.

It would be too much to claim that Sugarscape explains why wealth distributions are heavy-tailed, but the prevalence of inequality in variations of Sugarscape suggests that inequality is characteristic of many economies, even very simple ones. And experiments with rules that model taxation and other income transfers suggest that it is not easy to avoid or mitigate.

## 9.6 Implementing Sugarscape

Sugarscape is more complicated than the previous models, so I won't present the entire implementation here. I will outline the structure of the code and you can see the details in the Jupyter notebook for this chapter, `chap09.ipynb`, which is in the repository for this book. If you are not interested in the details, you can skip this section.

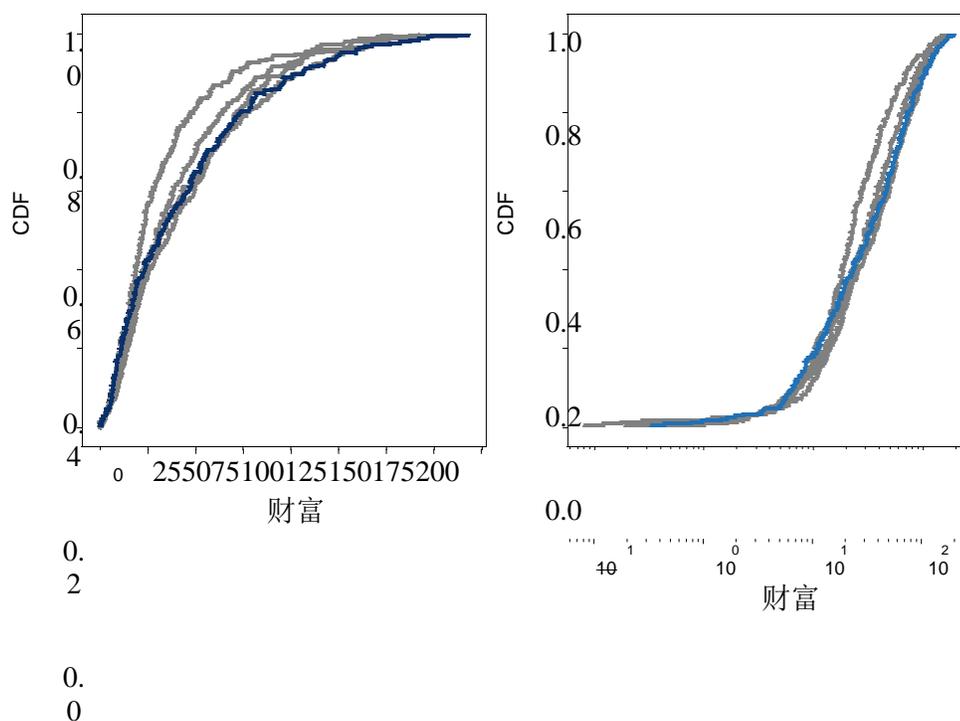


图 9.4: 糖(财富)在 100、200、300 和 400 级(灰线)和 500 级(暗线)之后的分配。线性刻度(左)和  $\log-x$  刻度(右)。

在对数尺度上，分布的形状类似于高斯分布或正态分布，尽管右尾被截断。如果在对数尺度上它是正常的，那么这个分布应该是对数正态的，也就是重尾分布。事实上，在几乎每个国家和世界上，财富的分配都是一个重尾分布。

如果说《甜景》解释了为什么财富分配是重尾分布，那就太过分了，但是《甜景》中各种不平等现象的普遍存在表明，不平等是许多经济体的特征，即使是非常简单的经济体也是如此。对税收和其他收入转移模式规则的试验表明，避免或减轻这些规则并不容易。

## 9.6 推行「糖景」

**Sugarscape** 比以前的模型更复杂，因此我不会在这里介绍整个实现。我将概述代码的结构，您可以在本章(chap09.ipynb)的 Jupyter 笔记本中看到详细信息，该章节在本书的资料库中。如果您对细节不感兴趣，可以跳过这一部分。

During each step, the agent moves, harvests sugar, and ages. Here is the `Agent` class and its `step` method:

```
class Agent:

    def step(self, env):
        self.loc = env.look_and_move(self.loc, self.vision)
        self.sugar += env.harvest(self.loc) - self.metabolism
        self.age += 1
```

The parameter `env` is a reference to the environment, which is a `Sugarscape` object. It provides methods `look_and_move` and `harvest`:

- `look_and_move` takes the location of the agent, which is a tuple of coordinates, and the range of the agent's vision, which is an integer. It returns the agent's new location, which is the visible cell with the most sugar.
- `harvest` takes the (new) location of the agent, and removes and returns the sugar at that location.

`Sugarscape` inherits from `Cell2D`, so it is similar to the other grid-based models we've seen.

The attributes include `agents`, which is a list of `Agent` objects, and `occupied`, which is a set of tuples, where each tuple contains the coordinates of a cell occupied by an agent.

Here is the `Sugarscape` class and its `step` method:

## 第九章基于 agent 的模型

---

在每个步骤中，代理移动、收获糖和年龄。下面是代理类及其步骤方法:

类别代理:

```
Def step (self, env) :
```

```
    这个词的意思是“自己”，意思是“自己”，即“自己”，即“自己”
```

参数 `env` 是对环境的引用，它是一个 `Sugarscape` 对象，它提供了 `look_and_move` 和 `harvest` 方法:

获取代理的位置，这是一个元组的坐标，以及代理的视野范围，这是一个整数。它返回代理的新位置，即具有最多糖。

`Harvest` 获取代理人的(新的)位置，然后移除并返回该位置的糖。

`Sugarscape` 继承自 `Cell2D`，因此它与我们见过的其他基于网格的模型类似。

这些属性包括代理(这是一个 `Agent` 对象列表)和 `occupation` (这是一组元组)，其中每个元组包含代理所占用的单元格的坐标。

下面是 `Sugarscape` 类及其步骤方法:

```
class Sugarscape(Cell2D):

    def step(self):

        # loop through the agents in random order
        random_order = np.random.permutation(self.agents)
        for agent in random_order:

            # mark the current cell unoccupied
            self.occupied.remove(agent.loc)

            # execute one step
            agent.step(self)

            # if the agent is dead, remove from the list
            if agent.is_starving():
                self.agents.remove(agent)
            else:
                # otherwise mark its cell occupied
                self.occupied.add(agent.loc)

        # grow back some sugar
        self.grow()
        return len(self.agents)
```

During each step, the `Sugarscape` uses the NumPy function `permutation` so it loops through the agents in random order. It invokes `step` on each agent and then checks whether it is dead. After all agents have moved, some of the sugar grows back. The return value is the number of agents still alive.

I won't show more details here; you can see them in the notebook for this chapter. If you want to learn more about NumPy, you might want to look at these functions in particular:

- `make_visible_locs`, which builds the array of locations an agent can see, depending on its vision. The locations are sorted by distance, with locations at the same distance appearing in random order. This function uses `np.random.shuffle` and `np.vstack`.



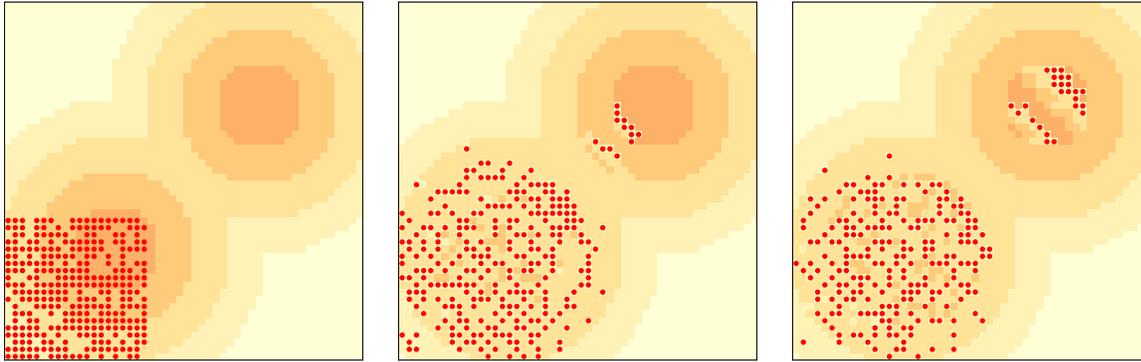


Figure 9.5: Wave behavior in Sugarscape: initial configuration (left), after 6 steps (middle) and after 12 steps (right).

- `make_capacity`, which initializes the capacity of the cells using NumPy functions `indices`, `hypot`, `minimum`, and `digitize`.
- `look_and_move`, which uses `argmax`.

## 9.7 Migration and Wave Behavior

Although the purpose of Sugarscape is not primarily to explore the movement of agents in space, Epstein and Axtell observed some interesting patterns when agents migrate.

If we start with all agents in the lower-left corner, they quickly move toward the closest “peak” of high-capacity cells. But if there are more agents than a single peak can support, they quickly exhaust the sugar and agents are forced to move into lower-capacity areas.

The ones with the longest vision cross the valley between the peaks and propagate toward the northeast in a pattern that resembles a wave front. Because

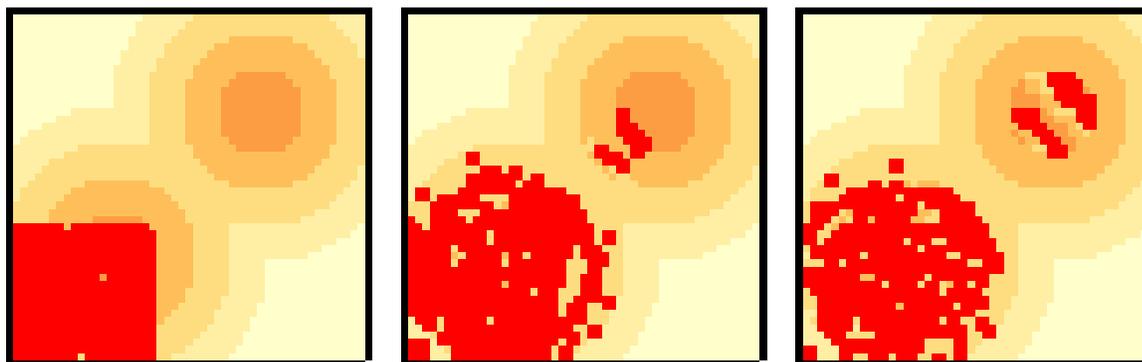


图 9.5: Sugarscape 中的 Wave 行为: 初始饱和(左)、6 步(中)和 12 步(右)。

使用 NumPy 初始化单元格的容量  
函数索引、假设值、最小值和数字化。

### 9.7 迁移和波浪行为

虽然《甜景》的主要目的不是探索行为体在空间中的移动，但爱泼斯坦和阿克斯特尔观察到了行为体移动时一些有趣的模式。

如果我们从左下角的所有代理开始，它们很快就会向最近的高容量细胞峰值移动。但是，如果有更多的代理商超过一个单一的高峰可以支持，他们很快耗尽糖和代理商被迫转移到低能力的地区。

那些拥有最长视野的人穿过山峰之间的山谷，向东北方向传播，其模式类似于波阵面。因为

they leave a stripe of empty cells behind them, other agents don't follow until the sugar grows back.

The result is a series of discrete waves of migration, where each wave resembles a coherent object, like the spaceships we saw in the Rule 110 CA and Game of Life (see Section 5.6 and Section 6.2).

Figure 9.5 shows the initial condition (left) and the state of the model after 6 steps (middle) and 12 steps (right). You can see the first two waves reaching and moving through the second peak, leaving a stripe of empty cells behind. You can see an animated version of this model, where the wave patterns are more clearly visible, in the notebook for this chapter.

These waves move diagonally, which is surprising because the agents themselves only move north or east, never northeast. Outcomes like this — groups or “aggregates” with properties and behaviors that the agents don't have — are common in agent-based models. We will see more examples in the next chapter.

## 9.8 Emergence

The examples in this chapter demonstrate one of the most important ideas in complexity science: emergence. An **emergent property** is a characteristic of a system that results from the interaction of its components, not from their properties.

To clarify what emergence is, it helps to consider what it isn't. For example, a brick wall is hard because bricks and mortar are hard, so that's not an emergent property. As another example, some rigid structures are built from flexible components, so that seems like a kind of emergence. But it is at best a weak kind, because structural properties follow from well understood laws of mechanics.

In contrast, the segregation we see in Schelling's model is an emergent property because it is not caused by racist agents. Even when the agents are only mildly xenophobic, the outcome of the system is substantially different from the intention of the agent's decisions.

它们留下一条空空的细胞，其他特工在糖重新长出之前不会跟进。

结果是一系列离散的移动波，每一个波都像一个连贯的物体，就像我们在 110 CA 法则和生命游戏中看到的宇宙飞船(见第 5.6 节和第 6.2 节)。

图 9.5 显示了初始条件(左)和 6 个步骤(中间)和 12 个步骤(右)之后的模型状态。你可以看到第一两个波达到并穿过第二个峰，留下一条空细胞带。你可以在本章的笔记本中看到这个模型的动画版本，其中的波形更加清晰可见。

这些波动沿对角线方向移动，这很令人惊讶，因为这些特工自己只向北或向东移动，从不向东北移动。像这样的结果 | 群体或聚合物的属性和行为的代理没有 | 是常见的基于代理的模型。我们将在下一章中看到更多的例子。

### 9.8 紧急情况

本章中的例子展示了复杂性科学中最重要的概念之一：涌现。突现属性是系统的一个特征，它来自组件之间的相互作用，而不是它们的属性。

为了弄清楚什么是涌现，我们需要考虑什么不是涌现。例如，砖墙是坚硬的，因为砖和灰泥是坚硬的，所以这不是一个突发性的属性。另一个例子是，一些刚性结构是由易弯曲的构件组成的，所以这看起来像是一种涌现。但它充其量只是一种弱力学性质，因为结构性质遵循众所周知的力学定律。

相比之下，我们在谢林的模型中看到的种族隔离是一种新兴的属性，因为它不是由种族主义因素引起的。即使这些代理人只是轻微的排外，这个系统的结果与代理人决策的意图大相径庭。

The distribution of wealth in Sugarscape might be an emergent property, but it is a weak example because we could reasonably predict it based on the distributions of vision, metabolism, and lifespan. The wave behavior we saw in the last example might be a stronger example, since the wave displays a capability — diagonal movement — that the agents do not have.

Emergent properties are surprising: it is hard to predict the behavior of the system even if we know all the rules. That difficulty is not an accident; in fact, it may be the defining characteristic of emergence.

As Wolfram discusses in *A New Kind of Science*, conventional science is based on the axiom that if you know the rules that govern a system, you can predict its behavior. What we call “laws” are often computational shortcuts that allow us to predict the outcome of a system without building or observing it.

But many cellular automata are **computationally irreducible**, which means that there are no shortcuts. The only way to get the outcome is to implement the system.

The same may be true of complex systems in general. For physical systems with more than a few components, there is usually no model that yields an analytic solution. Numerical methods provide a kind of computational shortcut, but there is still a qualitative difference.

Analytic solutions often provide a constant-time algorithm for prediction; that is, the run time of the computation does not depend on  $t$ , the time scale of prediction. But numerical methods, simulation, analog computation, and similar methods take time proportional to  $t$ . And for many systems, there is a bound on  $t$  beyond which we can't compute reliable predictions at all.

These observations suggest that emergent properties are fundamentally unpredictable, and that for complex systems we should not expect to find natural laws in the form of computational shortcuts.

To some people, “emergence” is another name for ignorance; by this reckoning, a property is emergent if we don't have a reductionist explanation for it, but if we come to understand it better in the future, it would no longer be emergent.

The status of emergent properties is a topic of debate, so it is appropriate to be skeptical. When we see an apparently emergent property, we should not

在 **Sugarscape**，财富的分配可能是一个突现的特性，但这是一个很弱的例子，因为我们可以根据视觉、新陈代谢和寿命的分布来合理地预测它。我们在上一个例子中看到的波动行为可能是一个更强的例子，因为波动显示了代理所没有的能力 | 对角线运动。

涌现的特性是令人惊讶的：即使我们知道所有的规则，也很难预测系统的行为。这种缺陷并非偶然，事实上，它可能是突现的本质特征。

正如 **Wolfram** 在《一种新的科学》中所讨论的，传统科学是基于这样一个公理：如果你知道支配一个系统的规则，你就可以预测它的行为。我们所说的定律”通常是计算的捷径，它允许我们不用建立或观察系统就能预测系统的结果。

但是许多细胞自动机在计算上是不可约的，这意味着没有捷径可走。得到结果的唯一方法是实现这个系统。

一般来说，复杂系统也可能是这样。对于具有多个组件的物理系统，通常不存在产生解析解的模型。数值方法提供了一种计算捷径，但仍然存在定性的差异。

解析解经常提供一个常数时间的预测算法，即计算的运行时间不依赖于  $t$ ，预测的时间尺度。但是数值方法、模拟、模拟计算和类似的方法都需要时间与  $t$  成正比。对于许多系统来说，有一个  $t$  的界限，超过这个界限我们根本无法计算出可靠的预测。

这些观察表明，涌现的性质是根本不可预测的，对于复杂的系统，我们不应该期望得到自然法则的计算捷径的形式。

对某些人来说，涌现是无知的另一个名称；根据这种推断，如果我们没有一个简化的解释，一个属性就是涌现，但是如果我们在未来更好地理解它，它就不再是涌现了。

涌现性质的状态是一个有争议的话题，因此持怀疑态度是合适的。当我们看到一个显然突现的性质时，我们不应该这样做

assume that there can never be a reductionist explanation. But neither should we assume that there has to be one.

The examples in this book and the principle of computational equivalence give good reasons to believe that at least some emergent properties can never be “explained” by a classical reductionist model.

You can read more about emergence at <http://thinkcomplex.com/emerge>.

## 9.9 Exercises

The code for this chapter is in the Jupyter notebook `chap09.ipynb` in the repository for this book. Open this notebook, read the code, and run the cells. You can use this notebook to work on the following exercises. My solutions are in `chap09soln.ipynb`.

**Exercise 9.1** Bill Bishop, author of *The Big Sort*, argues that American society is increasingly segregated by political opinion, as people choose to live among like-minded neighbors.

The mechanism Bishop hypothesizes is not that people, like the agents in Schelling’s model, are more likely to move if they are isolated, but that when they move for any reason, they are likely to choose a neighborhood with people like themselves.

Modify your implementation of Schelling’s model to simulate this kind of behavior and see if it yields similar degrees of segregation.

There are several ways you can model Bishop’s hypothesis. In my implementation, a random selection of agents moves during each step. Each agent considers  $k$  randomly-chosen empty locations and chooses the one with the highest fraction of similar neighbors. How does the degree of segregation depend on  $k$ ?

**Exercise 9.2** In the first version of SugarScape, we never add agents, so once the population falls, it never recovers. In the second version, we only replace agents when they die, so the population is constant. Now let’s see what happens if we add some “population pressure”.

假设从来不会有一个简化论的解释，但我们也不应该假设一定有一个。

本书中的例子和计算等价性原理给出了很好的理由相信，至少有一些突现性质是永远不能用经典的还原论模型来解释的。

你可以阅读更多关于涌现的 <http://thinkcomplex.com/emerge>。

### 9.9 练习

本章的代码在本书资料库中的 `chapyter` 笔记本 `chap09.ipynb` 中。打开这个笔记本，阅读代码，并运行单元格。你可以用这个笔记本做以下练习。我的解决方案在第 09 章 `soln.ipynb`。

练习 9.1 《大种类》的作者比尔·毕肖普认为，随着人们选择与志同道合的邻居生活在一起，美国社会正日益受到政治观点的隔离。

**Bishop** 假设的机制并不是人们，就像谢林模型中的代理人一样，如果他们被孤立，他们更有可能搬家，而是当他们因为任何原因搬家时，他们可能会选择一个和他们自己一样的人居住的社区。

修改谢林模型的实现，以模拟这种行为，并查看它是否会产生类似程度的隔离。

有几种方法可以模拟毕肖普的假设。在我的实现中，随机选择的代理在每个步骤中移动。每个代理考虑  $k$  个随机选择的空位置，然后选择相似邻居比例最高的位置。种族隔离的程度如何取决于  $k$ ？

练习 9.2 在第一个版本的 `SugarScape` 中，我们从不添加代理，因此一旦人口减少，它就永远不会恢复。在第二个版本中，我们只在病原体死亡时替换它们，因此人口是恒定的。现在让我们看看如果我们加上一些人口压力会发生什么。”。

Write a version of SugarScape that adds a new agent at the end of every step. Add code to compute the average vision and the average metabolism of the agents at the end of each step. Run the model for a few hundred steps and plot the population over time, as well as the average vision and average metabolism.

You should be able to implement this model by inheriting from `SugarScape` and overriding `__init__` and `step`.

**Exercise 9.3** Among people who study philosophy of mind, “Strong AI” is the theory that an appropriately-programmed computer could have a mind in the same sense that humans have minds.

John Searle presented a thought experiment called “The Chinese Room”, intended to show that Strong AI is false. You can read about it at <http://thinkcomplex.com/searle>.

What is the **system reply** to the Chinese Room argument? How does what you have learned about emergence influence your reaction to the system response?

## 第九章基于 agent 的模型

---

编写一个 `SugarScape` 版本，在每个步骤的末尾添加一个新代理。添加代码以计算每个步骤结束时的平均视力和代理的平均新陈代谢。将模型运行几百步，绘制一段时间内的人口，以及平均视力和平均新陈代谢情况。

您应该能够通过继承 `SugarScape` 并重写 `_init_` 和步骤来实现这个模型。

练习 9.3 在研究心灵哲学的人当中，“强人工智能”是这样一种理论，即一台经过适当编程的计算机可以拥有与人类拥有心灵同样意义上的心灵。

塞尔(johnsearle)提出了一个名为“中文房间”的思想实验，旨在证明强人工智能是错误的。你可以在 [http:// thinkcomplex.com/searle](http://thinkcomplex.com/searle) 上阅读相关文章。

系统如何回应中文房间的争论？你对突发事件的了解如何影响你对系统反应的反应？

# Chapter 10

## Herds, Flocks, and Traffic Jams

The agent-based models in the previous chapter are based on grids: the agents occupy discrete locations in two-dimensional space. In this chapter we consider agents that move in continuous space, including simulated cars on a one-dimensional highway and simulated birds in three-dimensional space.

The code for this chapter is in `chap10.ipynb`, which is a Jupyter notebook in the repository for this book. For more information about working with this code, see Section 0.3.

### 10.1 Traffic jams

What causes traffic jams? Sometimes there is an obvious cause, like an accident, a speed trap, or something else that disturbs the flow of traffic. But other times traffic jams appear for no apparent reason.

Agent-based models can help explain spontaneous traffic jams. As an example, I implement a highway simulation based on a model in Mitchell Resnick's book, *Turtles, Termites and Traffic Jams*.

Here's the class that represents the "highway":

## 第十章

### 牛群、羊群和果酱

前一章中的基于 `agent` 的模型是基于网格的: `agent` 占据了二维空间的离散位置。在本章中, 我们考虑连续空间移动的智能体, 包括一维高速公路上的模拟汽车和三维空间中的模拟鸟。

本章的代码在 `chap10.ipynb` 中, 这是本书资料库中的一个 `Jupyter` 笔记本。有关使用此代码的更多信息, 请参见 0.3 节。

#### 10.1 Traffic jams

什么原因导致交通堵塞? 有时会有一个明显的原因, 像事故, 速度陷阱, 或其他什么东西, 干扰了低温。但是有时候交通堵塞是没有明显的原因的。

基于 `agent` 的模型可以帮助解释自发的交通堵塞。作为一个例子, 我实现了一个基于 `Mitchell Resnick` 书中模型的高速公路模拟, 海龟、白蚁及交通拥堵。

下面是代表高速公路的类:

```
class Highway:

    def __init__(self, n=10, length=1000, eps=0):
        self.length = length
        self.eps = eps

        # create the drivers
        locs = np.linspace(0, length, n, endpoint=False)
        self.drivers = [Driver(loc) for loc in locs]

        # and link them up
        for i in range(n):
            j = (i+1) % n
            self.drivers[i].next = self.drivers[j]
```

`n` is the number of cars, `length` is the length of the highway, and `eps` is the amount of random noise we'll add to the system.

`locs` contains the locations of the drivers; the NumPy function `linspace` creates an array of `n` locations equally spaced between 0 and `length`.

The `drivers` attribute is a list of `Driver` objects. The `for` loop links them so each `Driver` contains a reference to the next. The highway is circular, so the last `Driver` contains a reference to the first.

During each time step, the `Highway` moves each of the drivers:

```
# Highway

def step(self):
    for driver in self.drivers:
        self.move(driver)
```

The `move` method lets the `Driver` choose its acceleration. Then `move` computes the updated speed and position. Here's the implementation:



```
# Highway

def move(self, driver):
    dist = self.distance(driver)

    # let the driver choose acceleration
    acc = driver.choose_acceleration(dist)
    acc = min(acc, self.max_acc)
    acc = max(acc, self.min_acc)
    speed = driver.speed + acc

    # add random noise to speed
    speed *= np.random.uniform(1-self.eps, 1+self.eps)

    # keep it nonnegative and under the speed limit
    speed = max(speed, 0)
    speed = min(speed, self.speed_limit)

    # if current speed would collide, stop
    if speed > dist:
        speed = 0

    # update speed and loc
    driver.speed = speed
    driver.loc += speed
```

`dist` is the distance between `driver` and the next driver ahead. This distance is passed to `choose_acceleration`, which specifies the behavior of the driver. This is the only decision the driver gets to make; everything else is determined by the “physics” of the simulation.

- `acc` is acceleration, which is bounded by `min_acc` and `max_acc`. In my implementation, cars can accelerate with `max_acc=1` and brake with `min_acc=-10`.
- `speed` is the old speed plus the requested acceleration, but then we make some adjustments. First, we add random noise to `speed`, because the world is not perfect. `eps` determines the magnitude of the relative error; for example, if `eps` is 0.02, `speed` is multiplied by a random factor between 0.98 and 1.02.

```
# 高速公路

移动(自己, 驱动程序):
    距离(驾驶员)

# 让司机选择加速度
选择 _ acceleration (dist) acc = min (acc, self.max
_ acc)
最大速度 = 驾驶速度 + 加速度

# 给速度加上随机噪音
速度 * = np.random.uniform (1-self. eps, 1 + self.eps)

# 保持非负速度, 低于限速速度 = 最大速度 0
速度 = 分钟(速度, 自己, 速度限制)

# 如果当前速度发生碰撞, 停止
如果 speed > dist:
    0

# 更新速度和 loc 驱动器
```

分区是驾驶员和前面的驾驶员之间的距离。这个距离被传递给选择 \_ 加速度, 这推测了驾驶员的行为。这是驱动程序做出的唯一决定; 其他一切都由模拟的物理学决定。

加速度是以 `min_acc` 和 `max_acc` 为界的。在我的实现中, 汽车可以加速与最大的 `acc = 1` 和刹车与

`Min_acc = -10`.

速度是原来的速度加上要求的加速度, 然后我们做一些调整。首先, 我们在速度中加入随机噪音, 因为世界并不完美。Eps 决定了相对误差的大小, 例如, 如果 eps 为 0.02, 速度乘以 0.98 到 1.02 之间的一个随机因子。

- Speed is bounded between 0 and `speed_limit`, which is 40 in my implementation, so cars are not allowed to go backward or speed.
- If the requested speed would cause a collision with the next car, `speed` is set to 0.
- Finally, we update the `speed` and `loc` attributes of `driver`.

Here's the definition for the `Driver` class:

```
class Driver:

    def __init__(self, loc, speed=0):
        self.loc = loc
        self.speed = speed

    def choose_acceleration(self, dist):
        return 1
```

The attributes `loc` and `speed` are the location and speed of the driver.

This implementation of `choose_acceleration` is simple: it always accelerates at the maximum rate.

Since the cars start out equally spaced, we expect them all to accelerate until they reach the speed limit, or until their speed exceeds the space between them. At that point, at least one “collision” will occur, causing some cars to stop.

Figure 10.1 shows a few steps in this process, starting with 30 cars and `eps=0.02`. On the left is the configuration after 16 time steps, with the highway mapped to a circle. Because of random noise, some cars are going faster than others, and the spacing has become uneven.

During the next time step (middle) there are two collisions, indicated by the triangles.

During the next time step (right) two cars collide with the stopped cars, and we can see the initial formation of a traffic jam. Once a jam forms, it tends to persist, with additional cars approaching from behind and colliding, and with cars in the front accelerating away.

Under some conditions, the jam itself propagates backwards, as you can see if you watch the animations in the notebook for this chapter.



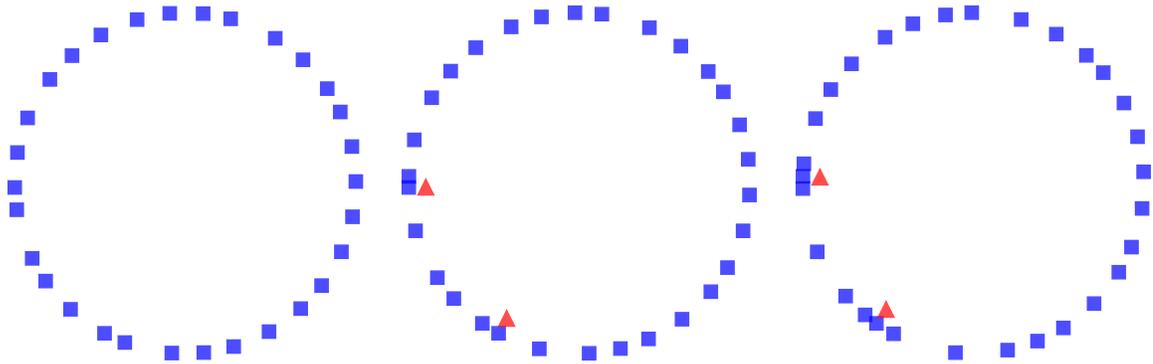


Figure 10.1: Simulation of drivers on a circular highway at three points in time. Squares indicate the position of the drivers; triangles indicate places where one driver has to brake to avoid another.

## 10.2 Random perturbation

As the number of cars increases, traffic jams become more severe. Figure 10.2 shows the average speed cars are able to achieve, as a function of the number of cars.

The top line shows results with  $\text{eps}=0$ , that is, with no random variation in speed. With 25 or fewer cars, the spacing between cars is greater than 40, which allows cars to reach and maintain the maximum speed, which is 40. With more than 25 cars, traffic jams form and the average speed drops quickly.

This effect is a direct result of the physics of the simulation, so it should not be surprising. If the length of the road is 1000, the spacing between  $n$  cars is  $1000/n$ . And since cars can't move faster than the space in front of them, the highest average speed we expect is  $1000/n$  or 40, whichever is less.

But that's the best case scenario. With just a small amount of randomness, things get much worse.

Figure 10.2 also shows results with  $\text{eps}=0.001$  and  $\text{eps}=0.01$ , which correspond to errors in speed of 0.1% and 1%.

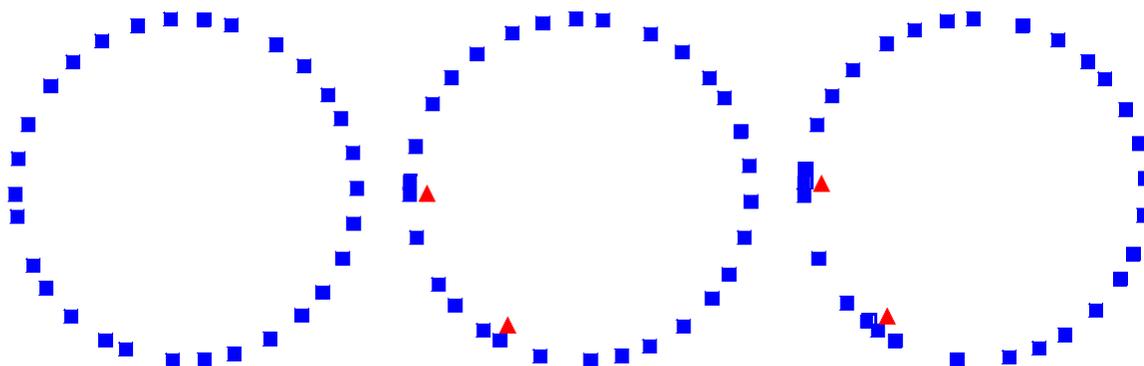


图 10.1: 环形高速公路上三个时间点的驾驶员模拟。正方形表示驾驶员的位置; 三角形表示一个驾驶员必须刹车以避免另一个驾驶员的位置。

### 10.2 随机摄动

随着汽车数量的增加, 交通堵塞变得越来越严重。图 10.2 显示了汽车的平均速度所能达到的, 作为汽车数量的函数。

上面一行显示  $\text{eps} = 0$  的结果, 也就是说, 在速度上没有任何随机变化。当车辆数量不超过 25 辆时, 车辆之间的间隔大于 40 辆, 这样车辆就可以达到并保持最大速度 40 辆。超过 25 辆车, 交通堵塞形成, 平均速度下降很快。

这个方面是模拟物理学的直接结果, 所以它不应该令人惊讶。如果道路的长度是 1000,  $n$  辆汽车之间的间距是  $1000/n$ 。由于汽车的速度不能超过它们前面的空间, 我们期望的最高平均速度是  $1000/n$  或 40, 以较低者为准。

但这是最好的情况, 只要有一点点随机性, 事情就会变得更糟。

图 10.2 还显示了  $\text{eps} = 0.001$  和  $\text{eps} = 0.01$  的结果, 这与速度误差 0.1% 和 1% 相对应。

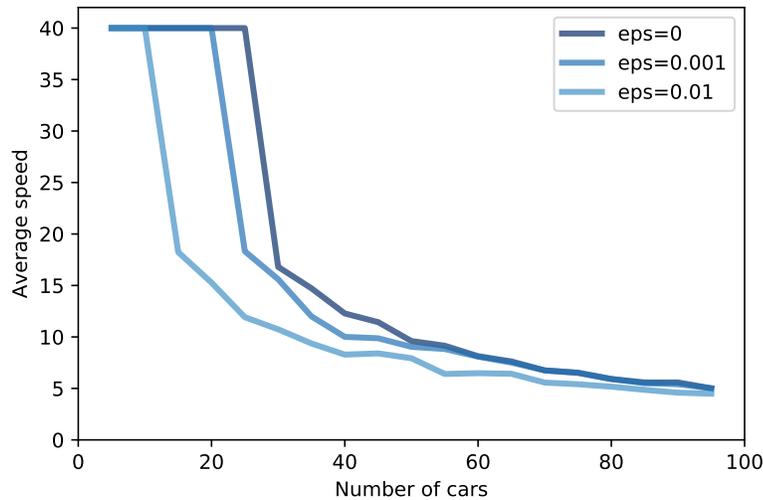


Figure 10.2: Average speed as a function of the number of cars, for three magnitudes of added random noise.

With 0.1% errors, the capacity of the highway drops from 25 to 20 (by “capacity” I mean the maximum number of cars that can reach and sustain the speed limit). And with 1% errors, the capacity drops to 10. Ugh.

As one of the exercises at the end of this chapter, you’ll have a chance to design a better driver; that is, you will experiment with different strategies in `choose_acceleration` and see if you can find driver behaviors that improve average speed.

### 10.3 Boids

In 1987 Craig Reynolds published “Flocks, herds and schools: A distributed behavioral model”, which describes an agent-based model of herd behavior. You can download his paper from <http://thinkcomplex.com/boid>.

Agents in this model are called “Boids”, which is both a contraction of “bird-oid” and an accented pronunciation of “bird” (although Boids are also used to model fish and herding land animals).

Each agent simulates three behaviors:

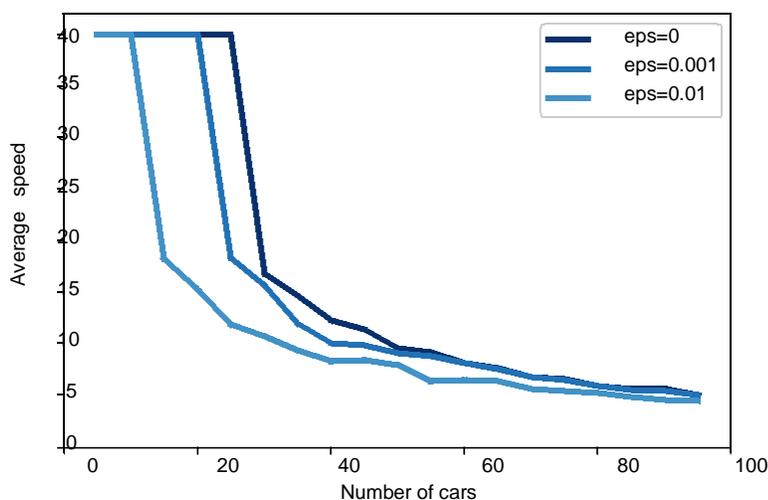


图 10.2: 平均车速作为车辆数量的函数，加上 3 个数量级的随机噪音。

由于 0.1% 的误差，高速公路的通行能力从 25 下降到 20(通行能力我指的是能够达到和维持速度限制的最大车辆数)。1% 的错误，容量下降到 10。呃。

作为本章最后的练习之一，你将有机会设计一个更好的驾驶员；也就是说，你将在选择加速方面尝试不同的策略，看看你能否找到提高平均速度的驾驶员行为。

### 10.3 Boids

1987 年 Craig Reynolds 发表了《羊群、牧群和学校: 分布式行为模型》，描述了个体为本模型的羊群行为。你可以从 <http://thinkcomplex.com/boid> 下载他的论文。

这个模型中的代理被称为“Boids”，它既是类鸟的缩写，也是鸟的重音发音(尽管 Boids 也用于建模 sh 和放牧陆地动物)。

每个代理模拟三种行为:

**Flock centering:** Move toward the center of the flock.

**Collision avoidance:** Avoid obstacles, including other Boids.

**Velocity matching:** Align velocity (speed and direction) with neighboring Boids.

Boids make decisions based on local information only; each Boid only sees (or pays attention to) other Boids in its field of vision.

In the repository for this book, you will find `Boids7.py`, which contains my implementation of Boids, based in part on the description in Gary William Flake's book, *The Computational Beauty of Nature*.

My implementation uses VPython, which is a library that provides 3-D graphics. VPython provides a `Vector` object, which I use to represent the position and velocity of Boids in three dimensions. You can read about them at <http://thinkcomplex.com/vector>.

## 10.4 The Boid algorithm

`Boids7.py` defines two classes: `Boid`, which implements the Boid behaviors, and `World`, which contains a list of Boids and a “carrot” the Boids are attracted to.

The `Boid` class defines the following methods:

- **center:** Finds other Boids within range and computes a vector toward their centroid.
- **avoid:** Finds objects, including other Boids, within a given range, and computes a vector that points away from their centroid.
- **align:** Finds other Boids within range and computes the average of their headings.
- **love:** Computes a vector that points toward the carrot.

群体向中心移动: 向岩石中心移动。

避免碰撞: 避免障碍物, 包括其他 Boids。

速度匹配: 调整速度(速度和方向)与邻近的 Boids。

博伊德人只根据当地信息做出决定; 每个博伊德人只看到(或注意到)其他博伊德人在其视野。

在这本书的资料库中, 你可以找到 `Boids7.py`, 它包含了我对 Boids 的实现, 部分基于加里·弗雷克的书《自然的计算之美》中的描述。

我的实现使用 `VPython`, 它是一个提供 3-D 图形的库。`VPython` 提供了一个 `Vector` 对象, 我用它来表示 Boids 在三维空间中的位置和速度。你可以在 <http://thinkcomplex.com/vector> 上读到他们的故事。

#### 10.4 Boid 算法

`Boids7.py` 中有两个类: `Boid`, 实现 Boid 行为; `World`, 包含 Boids 和一个 carrot 列表。

`Boid` 类包含以下方法:

找到范围内的其他 Boids 并计算向量  
他们的质心。

避免: 在给定范围内查找对象, 包括其他 Boids, 并计算远离其质心的向量。

找到范围内的其他 Boids 并计算他们的标题的平均值。

爱: 计算一个指向胡萝卜的向量。

Here's the implementation of `center`:

```
def center(self, boids, radius=1, angle=1):
    neighbors = self.get_neighbors(boids, radius, angle)
    vecs = [boid.pos for boid in neighbors]
    return self.vector_toward_center(vecs)
```

The parameters `radius` and `angle` are the radius and angle of the field of view, which determines which other Boids are taken into consideration. `radius` is in arbitrary units of length; `angle` is in radians.

`center` uses `get_neighbors` to get a list of Boid objects that are in the field of view. `vecs` is a list of Vector objects that represent their positions.

Finally, `vector_toward_center` computes a Vector that points from `self` to the centroid of neighbors.

Here's how `get_neighbors` works:

```
def get_neighbors(self, boids, radius, angle):
    neighbors = []
    for boid in boids:
        if boid is self:
            continue

        # if not in range, skip it
        offset = boid.pos - self.pos
        if offset.mag > radius:
            continue

        # if not within viewing angle, skip it
        if self.vel.diff_angle(offset) > angle:
            continue

        # otherwise add it to the list
        neighbors.append(boid)

    return neighbors
```

For each other Boid, `get_neighbors` uses vector subtraction to compute the

下面是 `center` 的实现:

```

Def center (self, boids, radius = 1, angle = 1):
    邻居 = self.get_邻居(boids, radius, angle)
    Vecs = [ boid.pos for boid in neighbors ]
    返回 self.vector_ towards_ center (vecs)

```

参数半径和角度是视场半径和角度，这决定了哪些其他的博伊德考虑。半径以任意长度单位表示，角度以弧度表示。

中心使用 `get_ neighbors` 来获取视野范围内的 `Boid` 对象列表。`Vecs` 是表示它们位置的 `Vector` 对象的列表。

最后，`Vector towards center` 计算出一个从 `self` 指向邻居质心的 `Vector`。

以下是如何得到邻居工作:

```

Def get_neighbors (self, boids, radius, angle):
    邻居[]
    对于大块头来说:
        如果大块头是自己:
            继续

        # 如果不在范围内, 就跳过它
        偏移 = boid.pos - self.pos
        if 偏移.mag > radius:
            继续

        # 如果不在可视角度内, 如果 self.vel.diff 角度(偏移量) > 角度:
            继续

        # 否则将其添加到列表邻居

    返回邻居

```

vector from `self` to `boid`. The magnitude of this vector is the distance between them; if this magnitude exceeds `radius`, we ignore `boid`.

`diff_angle` computes the angle between the velocity of `self`, which points in the direction the Boid is heading, and the position of `boid`. If this angle exceeds `angle`, we ignore `boid`.

Otherwise `boid` is in view, so we add it to `neighbors`.

Now here's the implementation of `vector_toward_center`, which computes a vector from `self` to the centroid of its neighbors.

```
def vector_toward_center(self, vecs):
    if vecs:
        center = np.mean(vecs)
        toward = vector(center - self.pos)
        return limit_vector(toward)
    else:
        return null_vector
```

VPython vectors work with NumPy, so `np.mean` computes the mean of `vecs`, which is a sequence of vectors. `limit_vector` limits the magnitude of the result to 1; `null_vector` has magnitude 0.

We use the same helper methods to implement `avoid`:

```
def avoid(self, boids, carrot, radius=0.3, angle=np.pi):
    objects = boids + [carrot]
    neighbors = self.get_neighbors(objects, radius, angle)
    vecs = [boid.pos for boid in neighbors]
    return -self.vector_toward_center(vecs)
```

`avoid` is similar to `center`, but it takes into account the carrot as well as the other Boids. Also, the parameters are different: `radius` is smaller, so Boids only avoid objects that are too close, and `angle` is wider, so Boids avoid objects from all directions. Finally, the result from `vector_toward_center` is negated, so it points *away* from the centroid of any objects that are too close.

Here's the implementation of `align`:

从自身到目标的矢量。这个矢量的大小就是它们之间的距离; 如果这个大小超过了半径, 我们就忽略玻色子。

差角计算的是自身速度, 指向博伊德的方向, 与博伊德的位置之间的角度。如果这个角度超过了角度, 我们就忽略波形。

否则视图中就是 boid, 因此我们将它添加到邻居中。

现在这里是 `vector towards center` 的实现, 它计算从 `self` 到其邻居的形心的向量。

```
Def vector _ towards _ center (self, vecs) :
```

```
    如果维克斯:
```

```
        Center = np.mean (vecs)
```

```
        Toward = vector (center-self.pos)
```

```
        返回限制 _ vector (朝向)
```

```
    其他:
```

```
        返回空向量
```

VPython 向量使用 NumPy, 因此 `np.mean` 计算向量的平均值, 即向量序列。将结果的值限制为 1; `null _ vector` 的值为 0。

我们使用相同的帮助器方法来实现避免:

```
Def avoid (self, boids, carrot, radius = 0.3, angle = np.pi) :
```

```
    Object = boids + [ carrot ]
```

```
    邻居 = self.get _ 邻居(对象, 半径, 角度)
```

```
    Vecs = [ boid.pos for boid in neighbors ]
```

```
    Return-self.vector _ towards _ center (vecs)
```

回避类似于中心, 但它考虑到了胡萝卜以及其他 Boids。此外, 参数不同: 半径较小, 因此 Boids 只避免物体过于接近, 角度较宽, 所以 Boids 避免物体从各个方向。最后, 向量向中心的结果是否定的, 所以它指向了离质心太近的物体。

下面是 `align` 的实现:

```
def align(self, boids, radius=0.5, angle=1):
    neighbors = self.get_neighbors(boids, radius, angle)
    vecs = [boid.vel for boid in neighbors]
    return self.vector_toward_center(vecs)
```

`align` is also similar to `center`; the big difference is that it computes the average of the neighbors' velocities, rather than their positions. If the neighbors point in a particular direction, the Boid tends to steer toward that direction.

Finally, `love` computes a vector that points in the direction of the carrot.

```
def love(self, carrot):
    toward = carrot.pos - self.pos
    return limit_vector(toward)
```

The results from `center`, `avoid`, `align`, and `love` are what Reynolds calls “acceleration requests”, where each request is intended to achieve a different goal.

## 10.5 Arbitration

To arbitrate among these possibly conflicting goals, we compute a weighted sum of the four requests:

```
def set_goal(self, boids, carrot):
    w_avoid = 10
    w_center = 3
    w_align = 1
    w_love = 10

    self.goal = (w_center * self.center(boids) +
                 w_avoid * self.avoid(boids, carrot) +
                 w_align * self.align(boids) +
                 w_love * self.love(carrot))
    self.goal.mag = 1
```

`w_center`, `w_avoid`, and the other weights determine the importance of the acceleration requests. Usually `w_avoid` is relatively high and `w_align` is relatively low.

```
def align(self, boids, radius=0.5, angle=1):
    neighbors = self.get_neighbors(boids, radius, angle)
    Vecs = [ boid.vel for boid in neighbors ]返回
    self.vector_towards_center (vecs)
```

`Align` 也类似于 `center`; 最大的不同之处在于它计算邻居速度的平均值, 而不是它们的位置。如果邻居指向一个特定的方向, 博伊德倾向于朝那个方向转向。

最后, 爱计算出一个指向胡萝卜方向的向量。

```
Def love (self, carrot):
    Toward = carrot.pos-self.pos
    返回限制_vector (朝向)
```

来自中心、避免、对齐和爱的结果就是雷诺兹所说的加速请求, 每个请求都是为了达到一个不同的目标。

### 10.5 仲裁

为了在这些可能的目标之间进行仲裁, 我们计算四个请求的加权和:

```
Def set_goal (self, boids, carrot):
    10
    3
    1
    爱 = 10

    = (w_center * self.center (boids) +
        避免 * self.avoid (boids, carrot) +
        W_align * self.align (boids) +
        爱自己, 爱(胡萝卜))
    1 = 1
```

和其他权重决定加速请求的重要性。通常 `w_avoid` 相对较高, `w_align` 相对较低。

After computing a goal for each Boid, we update their velocity and position:

```
def move(self, mu=0.1, dt=0.1):
    self.vel = (1-mu) * self.vel + mu * self.goal
    self.vel.mag = 1
    self.pos += dt * self.vel
    self.axis = self.length * self.vel
```

The new velocity is the weighted sum of the old velocity and the goal. The parameter `mu` determines how quickly the birds can change speed and direction. Then we normalize velocity so its magnitude is 1, and orient the axis of the Boid to point in the direction it is moving.

To update position, we multiply velocity by the time step, `dt`, to get the change in position. Finally, we update `axis` so the orientation of the Boid when it is drawn is aligned with its velocity.

Many parameters influence flock behavior, including the radius, angle and weight for each behavior, as well as maneuverability, `mu`. These parameters determine the ability of the Boids to form and maintain a flock, and the patterns of motion and organization within the flock. For some settings, the Boids resemble a flock of birds; other settings resemble a school of fish or a cloud of flying insects.

As one of the exercises at the end of this chapter, you can modify these parameters and see how they affect Boid behavior.

## 10.6 Emergence and free will

Many complex systems have properties, as a whole, that their components do not:

- The Rule 30 cellular automaton is deterministic, and the rules that govern its evolution are completely known. Nevertheless, it generates a sequence that is statistically indistinguishable from random.
- The agents in Schelling's model are not racist, but the outcome of their interactions is a high degree of segregation.

在计算每个 Boid 的目标之后，我们更新它们的速度和位置:

```
Def move (self, mu = 0.1, dt = 0.1):  
    (1-mu) * self.vel + mu * self.goal  
    1  
    = dt * self.vel  
    轴 = self.length * self.vel
```

新速度是旧速度和目标的加权和。参数 `mu` 决定了鸟类改变速度和方向的速度。然后将速度正态化，使它的大小为 1，并使波伊德的轴线指向它移动的方向。

为了更新位置，我们用时间步长 `dt` 乘以速度得到位置的变化。最后，我们更新轴，使博伊德的方向时，绘制是与其速度对齐。

结果表明，该系统具有良好的动力学性能，可以有效地提高系统的动力学性能。这些参数决定了博伊德人形成和维持锁的能力，以及锁内的运动模式和组织方式。在某些场景中，博伊德群岛就像一群鸟；其他场景则像一群鱼或一群飞鸟。

作为本章最后的练习之一，您可以修改这些参数，看看它们是如何影响大脑行为的。

## 10.6 涌现和自由意志

许多复杂系统作为一个整体具有其组成部分所没有的属性:

第 30 条细胞自动机是确定性的，并且支配其演变的规则是完全清楚的。然而，它产生了一个

在统计学上与随机无差别的序列。

谢林模型中的代理人不是种族主义者，但他们相互作用的结果是高度隔离。

- Agents in Sugarscape form waves that move diagonally even though the agents cannot.
- Traffic jams move backward even though the cars in them are moving forward.
- Flocks and herds behave as if they are centrally organized even though the animals in them are making individual decisions based on local information.

These examples suggest an approach to several old and challenging questions, including the problems of consciousness and free will.

Free will is the ability to make choices, but if our bodies and brains are governed by deterministic physical laws, our choices are completely determined.

Philosophers and scientists have proposed many possible resolutions to this apparent conflict; for example:

- William James proposed a two-stage model in which possible actions are generated by a random process and then selected by a deterministic process. In that case our actions are fundamentally unpredictable because the process that generates them includes a random element.
- David Hume suggested that our perception of making choices is an illusion; in that case, our actions are deterministic because the system that produces them is deterministic.

These arguments reconcile the conflict in opposite ways, but they agree that there is a conflict: the system cannot have free will if the parts are deterministic.

The complex systems in this book suggest the alternative that free will, at the level of options and decisions, is compatible with determinism at the level of neurons (or some lower level). In the same way that a traffic jam moves backward while the cars move forward, a person can have free will even though neurons don't.

Sugarscape 的代理商形成波浪，沿着对角线移动，即使特工们不能。

交通堵塞向后移动，即使车子在向前移动。

尽管其中的动物是根据当地信息做出个人决定的，但它们的行为却好像是集中组织的。

这些例子表明了一种方法，以几个古老的和具有挑战性的问题，包括问题的意识和自由意志。

自由意志是做出选择的能力，但是如果我们的身体和大脑受决定性的物理定律支配，我们的选择就完全被决定了。

哲学家和科学家已经提出了许多可能的解决方案，以解决这一明显的矛盾；例如：

威廉詹姆斯提出了一个两阶段的模型，其中可能的行动是由一个随机过程产生，然后选择一个确定性的过程。在这种情况下，我们的行为基本上是不可预测的，因为生成它们的过程包括一个随机元素。

大卫·休谟认为，我们做出选择的感知是一种错觉；在这种情况下，我们的行为是决定性的，因为产生这些行为的系统是决定性的。

这些争论以相反的方式调和了冲突，但是他们同意存在冲突：如果各个部分是确定的，系统就不可能有自由意志。

这本书中的复杂系统暗示了另一种选择，自由意志，在选择和决定的水平，是与决定论在神经元的水平(或一些更低的水平)兼容。就像汽车前进时交通阻塞向后移动一样，人可以拥有自由意志，即使神经元不能。

## 10.7 Exercises

The code for the traffic jam simulation is in the Jupyter notebook `chap09.ipynb` in the repository for this book. Open this notebook, read the code, and run the cells. You can use this notebook to work on the following exercise. My solutions are in `chap09soln.ipynb`.

**Exercise 10.1** In the traffic jam simulation, define a class, `BetterDriver`, that inherits from `Driver` and overrides `choose_acceleration`. See if you can define driving rules that do better than the basic implementation in `Driver`. You might try to achieve higher average speed, or a lower number of collisions.

**Exercise 10.2** The code for my Boid implementation is in `Boids7.py` in the repository for this book. To run it, you will need `VPython`, a library for 3-D graphics and animation. If you use Anaconda (as I recommend in Section 0.3), you can run the following in a terminal or Command Window:

```
conda install -c vpython vpython
```

Then run `Boids7.py`. It should either launch a browser or create a window in a running browser, and create an animated display showing Boids, as white cones, circling a red sphere, which is the carrot. If you click and move the mouse, you can move the carrot and see how the Boids react.

Read the code to see how the parameters control Boid behaviors. Experiment with different parameters. What happens if you “turn off” one of the behaviors by setting its weight to 0?

To generate more bird-like behavior, Flake suggests adding a behavior to maintain a clear line of sight; in other words, if there is another bird directly ahead, the Boid should move away laterally. What effect do you expect this rule to have on the behavior of the flock? Implement it and see.

**Exercise 10.3** Read more about free will at <http://thinkcomplex.com/will>. The view that free will is compatible with determinism is called **compatibilism**. One of the strongest challenges to compatibilism is the “consequence argument”. What is the consequence argument? What response can you give to the consequence argument based on what you have read in this book?

## 10.7 练习

Traffic jam 模拟的代码在本书资料库中的 Jupyter 笔记本 chap09.ipynb 中。打开这个笔记本，阅读代码，并运行单元格。你可以用这个笔记本做下面的练习。我的解决方案在第 09 章 soln.ipynb。

练习 10.1 在交通堵塞模拟中，一个类 BetterDriver 继承自 Driver 并重写选择 `_acceleration`。看看你能否找到比基本的驱动程序实现更好的驱动规则。你可以尝试达到更高的平均速度，或者更少的碰撞次数。

练习 10.2 我的 Boid 实现的代码在本书存储库中的 Boids7.py 中。要运行它，您需要 VPython，一个用于 3-D 图形和动画的库。如果你使用蟒蛇(正如我在 0.3 节中推荐的)，你可以在终端或命令窗口中运行以下命令：

```
Conda install-c vpython vpython
```

然后运行 Boids7.py。它应该启动一个浏览器或者在一个运行的浏览器中创建一个窗口，并且创建一个动画显示，显示 Boids，作为白色锥体，围绕一个红色的球体，这是胡萝卜。如果你点击并移动鼠标，你可以移动胡萝卜，看看博伊德如何反应。

阅读代码，了解参数如何控制 Boid 行为。不同参数的实验。如果你把其中一个行为的重量设置为 0，那会发生什么呢？

为了产生更多类似鸟的行为，弗莱克建议增加一个行为来保持视线清晰；换句话说，如果前面有另一只鸟，那么博伊德应该侧向移动。你认为这条规则对锁的行为有什么影响？实现它，然后看。

练习 10.3 阅读更多关于自由意志的 <http://thinkcomplex.com/>。认为自由意志与决定论相容的观点被称为共同主义。相容主义面临的巨大挑战之一是后果论”。什么是后果论证？根据你在这本书中读到的内容，你对结果论有什么反应？





# Chapter 11

## Evolution

The most important idea in biology, and possibly all of science, is the **theory of evolution by natural selection**, which claims that *new species are created and existing species change due to natural selection*. Natural selection is a process in which inherited variations between individuals cause differences in survival and reproduction.

Among people who know something about biology, the theory of evolution is widely regarded as a fact, which is to say that it is consistent with all current observations; it is highly unlikely to be contradicted by future observations; and, if it is revised in the future, the changes will almost certainly leave the central ideas substantially intact.

Nevertheless, many people do not believe in evolution. In a survey run by the Pew Research Center, survey respondents were asked which of the following claims is closer to their view:

1. Humans and other living things have evolved over time.
2. Humans and other living things have existed in their present form since the beginning of time.

About 34% of Americans chose the second (see <http://thinkcomplex.com/arda>).

## 第十一章

### 进化

生物学中最重要的思想，可能也是所有科学中最重要的思想，是通过自然选择进化的理论，该理论认为新物种的产生和现有物种的改变是由于自然选择。自然选择是一个个体间遗传变异导致生存和繁殖差异的过程。

在了解生物学的人中，进化论被广泛认为是一个事实，也就是说，它与当前的所有观察结果是一致的；它极不可能被未来的观察结果所否定；而且，如果在未来对它进行修改，几乎可以肯定的是，这些修改将使中心思想基本上完好无损。

然而，许多人并不相信进化论。在美国皮尤研究中心协会进行的一项调查中，调查对象被问及下列哪项声明更接近他们的观点：

1. 随着时间的推移，人类和其他生物进化了。
2. 人类和其他生物自时间开始以来就以现在的形式存在。

大约 34% 的美国人选择了第二种 <http://thinkcomplex.com/>。

Even among the ones who believe that living things have evolved, barely more than half believe that the cause of evolution is natural selection. In other words, only a third of Americans believe that the theory of evolution is true.

How is this possible? In my opinion, contributing factors include:

- Some people think that there is a conflict between evolution and their religious beliefs. Feeling like they have to reject one, they reject evolution.
- Others have been actively misinformed, often by members of the first group, so that much of what they know about evolution is misleading or false. For example, many people think that evolution means humans evolved from monkeys. It doesn't, and we didn't.
- And many people simply don't know anything about evolution.

There's probably not much I can do about the first group, but I think I can help the others. Empirically, the theory of evolution is hard for people to understand. At the same time, it is profoundly simple: for many people, once they understand it, it seems both obvious and irrefutable.

To help people make this transition from confusion to clarity, the most powerful tool I have found is computation. Ideas that are hard to understand in theory can be easy to understand when we see them happening in simulation. That is the goal of this chapter.

The code for this chapter is in `chap11.ipynb`, which is a Jupyter notebook in the repository for this book. For more information about working with this code, see Section 0.3.

## 11.1 Simulating evolution

I start with a simple model that demonstrates a basic form of evolution. According to the theory, the following features are sufficient to produce evolution:

- Replicators: We need a population of agents that can reproduce in some way. We'll start with replicators that make perfect copies of themselves. Later we'll add imperfect copying, that is, mutation.

即使是那些相信生物已经进化的人，也只有不到一半的人相信进化的原因是自然选择。换句话说，只有三分之一的美国人相信进化论是正确的。

这怎么可能呢？在我看来，促成因素包括：

有些人认为进化论和他们的宗教信仰之间存在矛盾。感觉他们必须拒绝一个，他们拒绝 `evolution` 的复数。

还有一些人被主动地误导了，通常是第一组的成员，所以他们对进化论的了解大多是误导性的或错误的。例如，许多人认为进化意味着人类从猴子进化而来。它不是，我们也不是。

许多人根本不知道进化论。

对于第一批人，我可能无能为力，但我想我可以帮助其他人。从经验上讲，进化论是很难让人理解的。与此同时，它却非常简单：对于许多人来说，一旦他们理解了它，它就显而易见，而且无可辩驳。

为了帮助人们从困惑过渡到清晰，我发现最强大的工具是计算。理论上难以理解的想法，当我们在模拟中看到它们发生时，就很容易理解。这就是本章的目的。

这一章的代码在 `chap1.ipynb` 中，这是本书资料库中的一个 `Jupyter` 笔记本。有关使用此代码的更多信息，请参见 0.3 节。

## 11.1 模拟进化

我从一个简单的模型开始，它演示了进化的基本形式。根据这一理论，以下特征足以产生进化：

复制因子：我们需要一群能够以某种方式繁殖的因子。我们将从复制子开始，它们会完美地复制自己。稍后我们会加上不完美的复制，也就是突变。

- Variation: We need variability in the population, that is, differences between individuals.
- Differential survival or reproduction: The differences between individuals have to affect their ability to survive or reproduce.

To simulate these features, we'll define a population of agents that represent individual organisms. Each agent has genetic information, called its **genotype**, which is the information that gets copied when the agent replicates. In our model<sup>1</sup>, a genotype is represented by a sequence of  $N$  binary digits (zeros and ones), where  $N$  is a parameter we choose.

To generate variation, we create a population with a variety of genotypes; later we will explore mechanisms that create or increase variation.

Finally, to generate differential survival and reproduction, we define a function that maps from each genotype to a **fitness**, where fitness is a quantity related to the ability of an agent to survive or reproduce.

## 11.2 Fitness landscape

The function that maps from genotype to fitness is called a **fitness landscape**. In the landscape metaphor, each genotype corresponds to a location in an  $N$ -dimensional space, and fitness corresponds to the “height” of the landscape at that location. For visualizations that might clarify this metaphor, see <http://thinkcomplex.com/fit>.

In biological terms, the fitness landscape represents information about how the genotype of an organism is related to its physical form and capabilities, called its **phenotype**, and how the phenotype interacts with its **environment**.

In the real world, fitness landscapes are complicated, but we don't need to build a realistic model. To induce evolution, we need *some* relationship between genotype and fitness, but it turns out that it can be *any* relationship. To demonstrate this point, we'll use a totally random fitness landscape.

---

<sup>1</sup>This model is a variant of the NK model developed primarily by Stuart Kauffman (see <http://thinkcomplex.com/nk>).

变异: 我们需要种群的变异性, 也就是说, 差异性个人之间。

生存或繁殖: 个体之间的差异取决于它们的生存或繁殖能力。

为了模拟这些特征, 我们将设计一个代表个体有机体的代理人群体。每个病原体都有遗传信息, 称为其基因型, 这是病原体复制时复制的信息。在我们的模型 1 中, 基因型由  $n$  个二进制数字(0 和 1)序列表示, 其中  $n$  是我们选择的一个参数。

为了产生变异, 我们创造了一个具有不同基因型的种群; 稍后我们将探索创造或增加变异的机制。

最后, 为了生成微生物的生存和繁殖, 我们设计了一个函数, 从每个基因型映射到一个特性, 其中特性是一个与特性生存或繁殖能力相关的数量。

## 11.2 健身景观

从基因型到因果关系映射的函数称为因果关系图。在景观隐喻中, 每个基因型对应于  $n$  维空间中的一个位置, 而后者对应于该位置景观的高度。有关可以澄清这个隐喻的可视化表达, 请参阅 [http:// thinkcomplex.com/fit](http://thinkcomplex.com/fit)。

从生物学的角度来看, 生物体的基因型与其物理形态和能力(称为表型)之间的关系, 以及表型与环境之间的相互作用。

在现实世界中, 空间景观是复杂的, 但我们不需要建立一个真实的模型。为了诱导进化, 我们需要在基因型和因果关系之间建立某种关系, 但结果表明, 它可以是任何关系。为了演示这一点, 我们将使用一个完全随机的景观。

---

<sup>1</sup> 这个模型是 NK 模型的一个变体, 这个模型主要是由 Stuart Kau man 开发的(见 <http://thinkcomplex.com/NK>)。

Here is the definition for a class that represents a fitness landscape:

```
class FitnessLandscape:

    def __init__(self, N):
        self.N = N
        self.one_values = np.random.random(N)
        self.zero_values = np.random.random(N)

    def fitness(self, loc):
        fs = np.where(loc, self.one_values,
                     self.zero_values)
        return fs.mean()
```

The genotype of an agent, which corresponds to its location in the fitness landscape, is represented by a NumPy array of zeros and ones called `loc`. The fitness of a given genotype is the mean of  $N$  **fitness contributions**, one for each element of `loc`.

To compute the fitness of a genotype, `FitnessLandscape` uses two arrays: `one_values`, which contains the fitness contributions of having a 1 in each element of `loc`, and `zero_values`, which contains the fitness contributions of having a 0.

The `fitness` method uses `np.where` to select a value from `one_values` where `loc` has a 1, and a value from `zero_values` where `loc` has a 0.

As an example, suppose  $N=3$  and

```
one_values = [0.1, 0.2, 0.3]
zero_values = [0.4, 0.7, 0.9]
```

In that case, the fitness of `loc = [0, 1, 0]` would be the mean of `[0.4, 0.2, 0.9]`, which is 0.5.

## 11.3 Agents

Next we need agents. Here's the class definition:

下面是一个代表空间景观的类的描述:

类别适宜/景观:

返回文章页面【一分钟科普】:

`N = n`

一个值 = `np.random.random (n)`

`0 _ values = np.random.random (n)`

Def fitness (self, loc) :

其中(loc, self.one \_ values,  
0 值)

返回 `fs.mean ()`

代理的基因型(与其在前景中的位置相对应)由一个由 0 和 1 组成的 NumPy 数组表示, 该数组称为 loc。给定基因型的特性是 n 贡献的平均值, 每一个 loc 元素都有一个特性。

为了计算基因型的性能, FitnessLandscape 使用两个数组: one \_ values, 它包含 loc 中每个元素的性能贡献为 1, zero \_ values, 它包含具有 0 的性能贡献为 0。

适应度方法使用 `np.where` 从一个值中选择一个值, 其中 loc 为 1, 从零个值中选择一个值, 其中 loc 为 0。

举个例子, 假设 `n = 3`

一个值 = `[0.1,0.2,0.3]`

`0 _ values = [0.4,0.7,0.9]`

在这种情况下, `loc = [0,1,0]`的平均值为`[0.4,0.2,0.9]`, 也就是 0.5。

### 11.3 代理商

接下来我们需要特工, 这是班级划分:

```
class Agent:

    def __init__(self, loc, fit_land):
        self.loc = loc
        self.fit_land = fit_land
        self.fitness = fit_land.fitness(self.loc)

    def copy(self):
        return Agent(self.loc, self.fit_land)
```

The attributes of an `Agent` are:

- `loc`: The location of the `Agent` in the fitness landscape.
- `fit_land`: A reference to a `FitnessLandscape` object.
- `fitness`: The fitness of this `Agent` in the `FitnessLandscape`, represented as a number between 0 and 1.

`Agent` provides `copy`, which copies the genotype exactly. Later, we will see a version that copies with mutation, but mutation is not necessary for evolution.

## 11.4 Simulation

Now that we have agents and a fitness landscape, I'll define a class called `Simulation` that simulates the creation, reproduction, and death of the agents. To avoid getting bogged down, I'll present a simplified version of the code here; you can see the details in the notebook for this chapter.

Here's the definition of `Simulation`:

```
class Simulation:

    def __init__(self, fit_land, agents):
        self.fit_land = fit_land
        self.agents = agents
```

The attributes of a `Simulation` are:

类别代理:

返回文章页面【一分钟科普】:

自我介绍

自己, 适合地, 适合地

健康 = 健康的土地

Def copy (self) :

返回代理人(self.loc, self.fit\_land)

Agent 的属性如下:

主体在空间景观中的位置。

适合土地: 一个适合景观对象的参考。

适应性: 这个 Agent 在适应性景观中的适应性, 表示为 0 到 1 之间的数字。

代理商提供复制品, 准确复制基因型。后来, 我们会看到一个带有突变的复制版本, 但是突变对于进化来说是不必要的。

## 11.4 模拟

现在我们已经有了代理和前景, 我将设计一个名为模拟的类, 模拟代理的创建、复制和死亡。为了避免陷入困境, 我将在这里展示一个简化版本的代码; 您可以在本章的笔记本中看到详细信息。

下面是模拟的概念:

类模拟:

返回文章页面【一分钟科普】:

自己, 适合地, 适合地

自己, 代理人, 代理人

- `fit_land`: A reference to a `FitnessLandscape` object.
- `agents`: An array of `Agent` objects.

The most important function in `Simulation` is `step`, which simulates one time step:

```
# class Simulation:

    def step(self):
        n = len(self.agents)
        fits = self.get_fitnesses()

        # see who dies
        index_dead = self.choose_dead(fits)
        num_dead = len(index_dead)

        # replace the dead with copies of the living
        replacements = self.choose_replacements(num_dead, fits)
        self.agents[index_dead] = replacements
```

`step` uses three other methods:

- `get_fitnesses` returns an array containing the fitness of each agent.
- `choose_dead` decides which agents die during this time step, and returns an array that contains the indices of the dead agents.
- `choose_replacements` decides which agents reproduce during this time step, invokes `copy` on each one, and returns an array of new `Agent` objects.

In this version of the simulation, the number of new agents during each time step equals the number of dead agents, so the number of live agents is constant.

## 11.5 No differentiation

Before we run the simulation, we have to specify the behavior of `choose_dead` and `choose_replacements`. We'll start with simple versions of these functions that don't depend on fitness:



```
# class Simulation

    def choose_dead(self, fits):
        n = len(self.agents)
        is_dead = np.random.random(n) < 0.1
        index_dead = np.nonzero(is_dead)[0]
        return index_dead
```

In `choose_dead`, `n` is the number of agents and `is_dead` is a boolean array that contains `True` for the agents who die during this time step. In this version, every agent has the same probability of dying: 0.1. `choose_dead` uses `np.nonzero` to find the indices of the non-zero elements of `is_dead` (`True` is considered non-zero).

```
# class Simulation

    def choose_replacements(self, n, fits):
        agents = np.random.choice(self.agents, size=n, replace=True)
        replacements = [agent.copy() for agent in agents]
        return replacements
```

In `choose_replacements`, `n` is the number of agents who reproduce during this time step. It uses `np.random.choice` to choose `n` agents with replacement. Then it invokes `copy` on each one and returns a list of new `Agent` objects.

These methods don't depend on fitness, so this simulation does not have differential survival or reproduction. As a result, we should not expect to see evolution. But how can we tell?

## 11.6 Evidence of evolution

The most inclusive definition of evolution is a change in the distribution of genotypes in a population. Evolution is an aggregate effect: in other words, individuals don't evolve; populations do.

In this simulation, genotypes are locations in a high-dimensional space, so it is hard to visualize changes in their distribution. However, if the genotypes change, we expect their fitness to change as well. So we will use *changes in*



*the distribution of fitness* as evidence of evolution. In particular, we'll look at the mean and standard deviation of fitness over time.

Before we run the simulation, we have to add an `Instrument`, which is an object that gets updated after each time step, computes a statistic of interest, or “metric”, and stores the result in a sequence we can plot later.

Here is the parent class for all instruments:

```
class Instrument:
    def __init__(self):
        self.metrics = []
```

And here's the definition for `MeanFitness`, an instrument that computes the mean fitness of the population at each time step:

```
class MeanFitness(Instrument):
    def update(self, sim):
        mean = np.nanmean(sim.get_fitnesses())
        self.metrics.append(mean)
```

Now we're ready to run the simulation. To avoid the effect of random changes in the starting population, we start every simulation with the same set of agents. And to make sure we explore the entire fitness landscape, we start with one agent at every location. Here's the code that creates the `Simulation`:

```
N = 8
fit_land = FitnessLandscape(N)
agents = make_all_agents(fit_land, Agent)
sim = Simulation(fit_land, agents)
```

`make_all_agents` creates one `Agent` for every location; the implementation is in the notebook for this chapter.

Now we can create and add a `MeanFitness` instrument, run the simulation, and plot the results:

```
instrument = MeanFitness()
sim.add_instrument(instrument)
sim.run()
```

作为进化证据的适度分布。特别是，我们将看看随着时间推移的平均值和标准差。

在我们运行模拟之前，我们必须添加一个 **Instrument**，它是一个对象，在每个时间步骤之后得到更新，计算感兴趣的统计数据，或度量”，并将结果存储在一个序列中，我们以后可以绘制。

以下是所有乐器的父类:

```
类别乐器:
```

```
    返回文章页面自我实现译者:
```

```
    []
```

下面是 **MeanFitness** 的定义，它是一个在每个时间步计算总体平均值的工具:

```
    课程名称健体(乐器):
```

```
    Def update (self, sim):
```

```
        Mean = np.nanmean (sim.get _ fitness ())
```

```
        附加(平均)
```

现在我们可以开始模拟了。为了避免起始种群随机变化的影响，我们用相同的代理集合开始每个模拟。为了确保我们能够探索整个领域，我们从每个地点都有一个代理商开始。下面是创建模拟的代码:

```
N = 8
```

```
适合土地 = 适合景观(n)
```

```
所有的代理(适合土地, 代理)
```

```
Sim = 模拟(fit _ land, agent)
```

为每个位置创建一个 **Agent**; 实现在本章的笔记本中。

现在我们可以创建并添加一个 **MeanFitness** 工具，运行模拟，并绘制结果:

```
乐器 = MeanFitness ()
```

```
Add _ instrument (instrument)
```

```
Sim.run ()
```

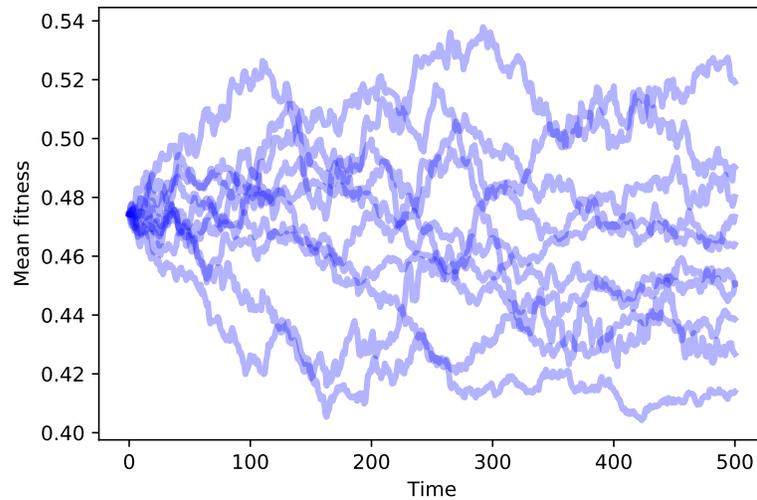


Figure 11.1: Mean fitness over time for 10 simulations with no differential survival or reproduction.

`Simulation` keeps a list of `Instrument` objects. After each time step it invokes `update` on each `Instrument` in the list.

Figure 11.1 shows the result of running this simulation 10 times. The mean fitness of the population drifts up or down at random. Since the distribution of fitness changes over time, we infer that the distribution of phenotypes is also changing. By the most inclusive definition, this **random walk** is a kind of evolution. But it is not a particularly interesting kind.

In particular, this kind of evolution does not explain how biological species change over time, or how new species appear. The theory of evolution is powerful because it explains phenomena we see in the natural world that seem inexplicable:

- **Adaptation:** Species interact with their environments in ways that seem too complex, too intricate, and too clever to happen by chance. Many features of natural systems seem as if they were designed.
- **Increasing diversity:** Over time the number of species on earth has generally increased (despite several periods of mass extinction).

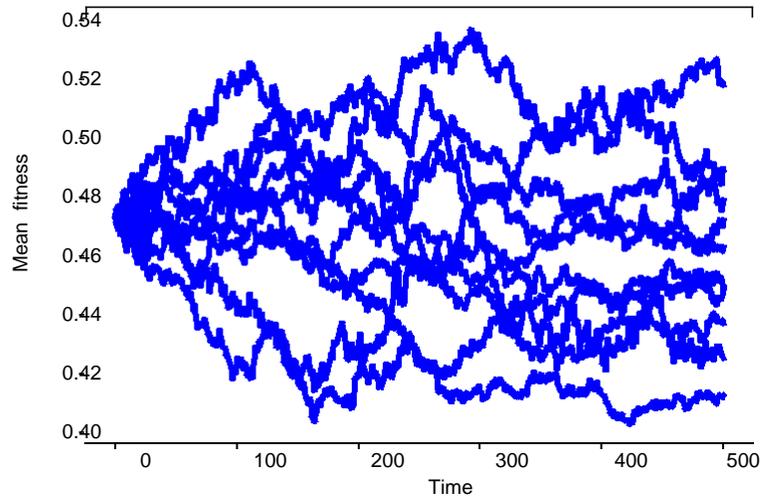


图 11.1:10 个模拟的平均值随时间的变化，没有指向性生存或繁殖。

模拟保存了一个仪器对象列表。在每个时间步骤之后，它调用更新列表中的每个仪器。

图 11.1 显示了运行这个模拟 10 次的结果。人口的平均值随机上升或下降。由于表型的分布随时间而变化，我们推断表型的分布也在变化。从最具包容性的角度来看，这种随机游走是一种进化。但这并不是一种特别有趣的方式。

特别是，这种进化并不能解释生物物种是如何随着时间变化的，或者新物种是如何出现的。进化论之所以强大，是因为它解释了我们在自然界中看到的令人费解的现象：

适应性: 物种与环境的互动方式似乎太复杂，太复杂，太聪明，不可能偶然发生。很多

自然系统的特征似乎是。

增加物种多样性: 随着时间的推移，地球上物种的数量普遍增加(尽管有几次大规模灭绝)。

- Increasing complexity: The history of life on earth starts with relatively simple life forms, with more complex organisms appearing later in the geological record.

These are the phenomena we want to explain. So far, our model doesn't do the job.

## 11.7 Differential survival

Let's add one more ingredient, differential survival. Here's a class that extends `Simulation` and overrides `choose_dead`:

```
class SimWithDiffSurvival(Simulation):  
  
    def choose_dead(self, fits):  
        n = len(self.agents)  
        is_dead = np.random.random(n) > fits  
        index_dead = np.nonzero(is_dead)[0]  
        return index_dead
```

Now the probability of survival depends on fitness; in fact, in this version, the probability that an agent survives each time step *is* its fitness.

Since agents with low fitness are more likely to die, agents with high fitness are more likely to survive long enough to reproduce. Over time we expect the number of low-fitness agents to decrease, and the number of high-fitness agents to increase.

Figure 11.2 shows mean fitness over time for 10 simulations with differential survival. Mean fitness increases quickly at first, but then levels off.

You can probably figure out why it levels off: if there is only one agent at a particular location and it dies, it leaves that location unoccupied. Without mutation, there is no way for it to be occupied again.

With  $N=8$ , this simulation starts with 256 agents occupying all possible locations. Over time, the number of occupied locations decreases; if the simulation runs long enough, eventually all agents will occupy the same location.



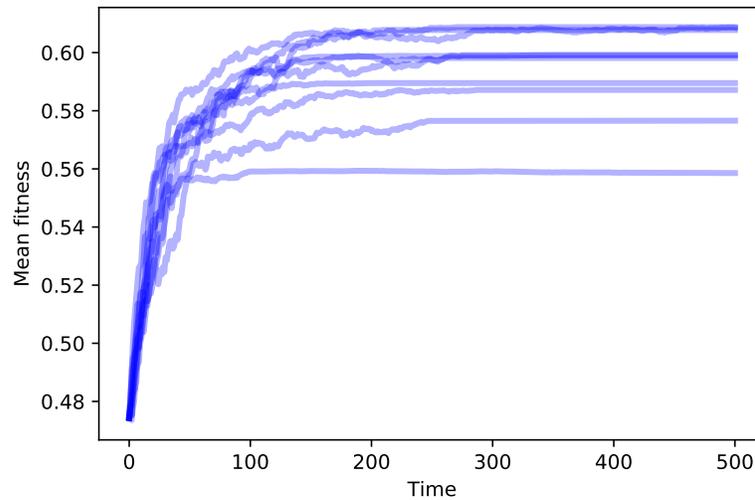


Figure 11.2: Mean fitness over time for 10 simulations with differential survival.

So this simulation starts to explain adaptation: increasing fitness means that the species is getting better at surviving in its environment. But the number of occupied locations decreases over time, so this model does not explain increasing diversity at all.

In the notebook for this chapter, you will see the effect of differential reproduction. As you might expect, differential reproduction also increases mean fitness. But without mutation, we still don't see increasing diversity.

## 11.8 Mutation

In the simulations so far, we start with the maximum possible diversity — one agent at every location in the landscape — and end with the minimum possible diversity, all agents at one location.

That's almost the opposite of what happened in the natural world, which apparently began with a single species that branched, over time, into the millions, or possibly billions, of species on Earth today (see <http://thinkcomplex.com/bio>).

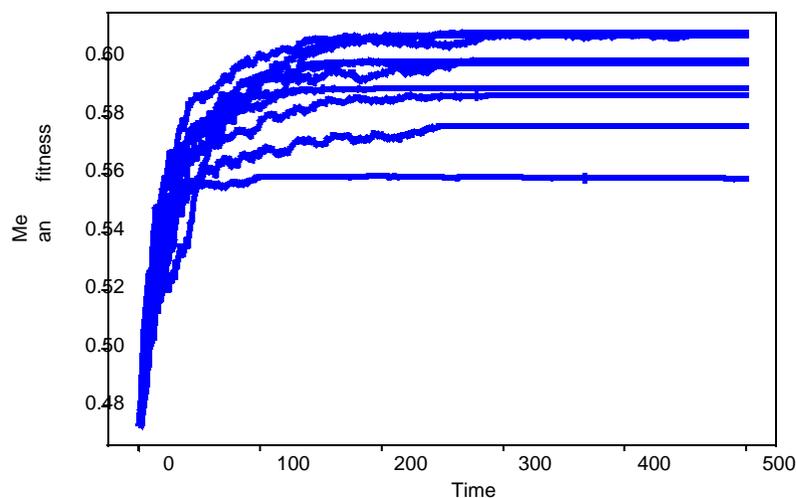


图 11.2:10 次分级生存模拟的随时间变化的平均值。

所以这个模拟开始解释适应性: 增长意味着物种在它的环境中生存得越来越好。但是占用位置的数量随着时间的推移而减少, 所以这个模型根本不能解释增加的多样性。

在这一章的笔记本中, 你会看到参照复制的方面。正如你可能预期的那样, 参数复制也增加了平均值。但是如果没有突变, 我们仍然看不到增加的多样性。

## 11.8 变异

在到目前为止的模拟中, 我们从最大可能的多样性开始 | 景观中的每个位置都有一个代理 | 最小可能的多样性结束, 所有代理都在一个位置。

这几乎与自然界发生的情况相反, 自然界显然是从一个单一的物种开始的, 随着时间的推移, 分支到今天地球上数百万, 甚至可能是数十亿个物种的 <http://thinkcomplex.com/bio>。

With perfect copying in our model, we never see increasing diversity. But if we add mutation, along with differential survival and reproduction, we get a step closer to understanding evolution in nature.

Here is a class definition that extends `Agent` and overrides `copy`:

```
class Mutant(Agent):  
  
    def copy(self, probab_mutate=0.05)::  
        if np.random.random() > probab_mutate:  
            loc = self.loc.copy()  
        else:  
            direction = np.random.randint(self.fit_land.N)  
            loc = self.mutate(direction)  
        return Mutant(loc, self.fit_land)
```

In this model of mutation, every time we call `copy`, there is a 5% chance of mutation. In case of mutation, we choose a random direction from the current location — that is, a random bit in the genotype — and flip it. Here’s `mutate`:

```
def mutate(self, direction):  
    new_loc = self.loc.copy()  
    new_loc[direction] ^= 1  
    return new_loc
```

The operator `^=` computes “exclusive OR”; with the operand 1, it has the effect of flipping a bit (see <http://thinkcomplex.com/xor>).

Now that we have mutation, we don’t have to start with an agent at every location. Instead, we can start with the minimum variability: all agents at the same location.

Figure 11.3 shows the results of 10 simulations with mutation and differential survival and reproduction. In every case, the population evolves toward the location with maximum fitness.

To measure diversity in the population, we can plot the number of occupied locations after each time step. Figure 11.4 shows the results. We start with 100 agents at the same location. As mutations occur, the number of occupied locations increases quickly.

在我们的模型中完美的复制，我们从来没有看到增加的多样性。但是如果我们再加上基因突变，以及不同生物的生存和繁殖，我们就离理解自然界的进化更近了一步。

下面是一个扩展 Agent 并覆盖 copy 的类命令：

```
变种人(代理人):

    返回文章页面防御拷贝(self, probab _ mutate = 0.05) ::
        如果 np.random.random () > probab _ mutate:
            Loc = self.loc.copy ()
        其他:
            方向 = np.rando.int (self.fit _ land.N)
            (方向)
        返回突变体(loc, self.fit _ land)
```

在这个变异模型中，每次我们称之为复制，就有 5% 的机会发生变异。在突变的情况下，我们从当前位置 | 选择一个随机方向，也就是说，在基因型 | 中选择一个随机位并激发它。下面是变异：

```
Def mutate (self, direction) :
    New _ loc = self.loc.copy ()
    New _ loc [ direction ] ^ = 1
    返回 new loc
```

运算符  $\wedge =$  计算独占或”；对于操作数 1，它具有删除位的 <http://thinkcomplex.com/xor>。

既然我们已经有了变异，我们就不必在每个地方都有一个变异体。相反，我们可以从最小的可变性开始：所有的代理人在同一个位置。

图 11.3 显示了 10 个模拟突变和双向生存和繁殖的结果。在每一种情况下，种群都以最大的优势向位置方向演化。

为了测量种群的多样性，我们可以在每个时间步骤之后绘制被占据的位置数量。图 11.4 显示了结果。我们从同一地点的 100 名特工开始。随着突变的发生，占据位置的数量迅速增加。

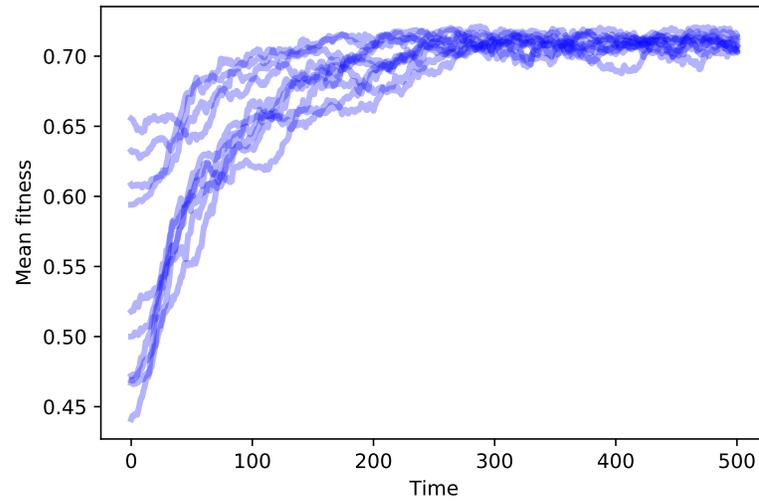


Figure 11.3: Mean fitness over time for 10 simulations with mutation and differential survival and reproduction.

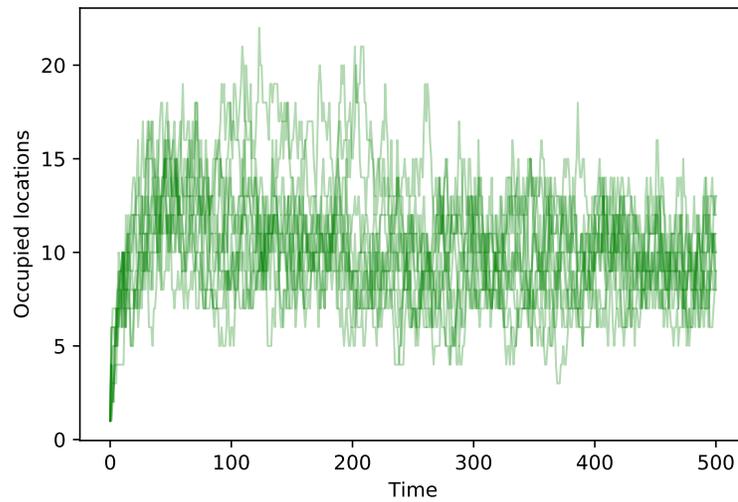


Figure 11.4: Number of occupied locations over time for 10 simulations with mutation and differential survival and reproduction.

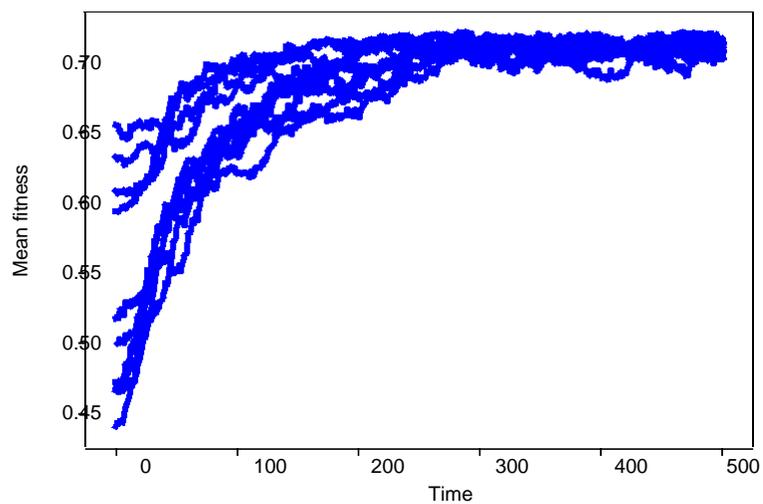


图 11.3:10 个模拟突变和差异生存和繁殖的平均值随时间的变化。

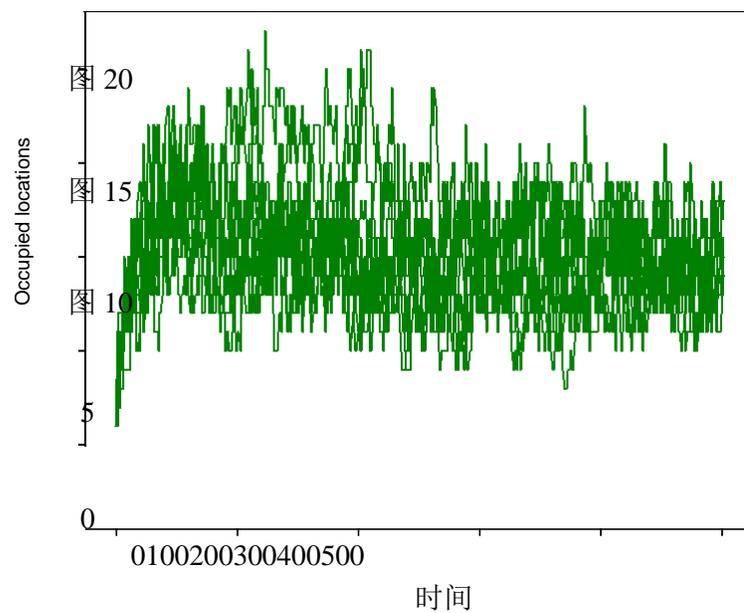


图 11.4: 随着时间的推移, 10 个模拟突变和二级生存和繁殖的占用位置的数量。

When an agent discovers a high-fitness location, it is more likely to survive and reproduce. Agents at lower-fitness locations eventually die out. Over time, the population migrates through the landscape until most agents are at the location with the highest fitness.

At that point, the system reaches an equilibrium where mutation occupies new locations at the same rate that differential survival causes lower-fitness locations to be left empty.

The number of occupied locations in equilibrium depends on the mutation rate and the degree of differential survival. In these simulations the number of unique occupied locations at any point is typically 5–15.

It is important to remember that the agents in this model don't move, just as the genotype of an organism doesn't change. When an agent dies, it can leave a location unoccupied. And when a mutation occurs, it can occupy a new location. As agents disappear from some locations and appear in others, the population migrates across the landscape, like a glider in Game of Life. But organisms don't evolve; populations do.

## 11.9 Speciation

The theory of evolution says that natural selection changes existing species and creates new ones. In our model, we have seen changes, but we have not seen a new species. It's not even clear, in the model, what a new species would look like.

Among species that reproduce sexually, two organisms are considered the same species if they can breed and produce fertile offspring. But the agents in the model don't reproduce sexually, so this definition doesn't apply.

Among organisms that reproduce asexually, like bacteria, the definition of species is not as clear-cut. Generally, a population is considered a species if their genotypes form a cluster, that is, if the genetic differences within the population are small compared to the differences between populations.

Before we can model new species, we need the ability to identify clusters of agents in the landscape, which means we need a definition of **distance** between

当一个代理人发现一个高空位置，它更有可能生存和繁殖。低温环境下的病原体最终会灭绝。随着时间的推移，种群在地形中迁移，直到大多数代理人到达最高的位置。

在这一点上，系统达到了一个平衡，突变以同样的速度占据新的位置，同样的速度下降，导致下降位置空置。

平衡点的数量取决于突变率和双参数生存的程度。在这些模拟中，任何一点的唯一占用位置的数量通常为 5{15}。

重要的是要记住，这个模型中的作用者不会移动，就像一个生物体的基因型不会改变一样。当一个代理人死了，它可以留下一个无人居住的地方。当突变发生时，它会占据一个新的位置。当代理人从某些地方消失而出现在另一些地方时，人口就会在这片土地上迁移，就像《生命游戏》中的滑翔机。但是有机体并不进化，而是种群进化。

### 11.9 物种形成

进化论认为自然选择改变了现有的物种并创造了新的物种。在我们的模型中，我们看到了变化，但是我们没有看到一个新的物种。在这个模型中，一个新物种会是什么样子，甚至还不清楚。

在有性繁殖的物种中，如果两种有机体能够繁殖并产生可育的春天，那么它们就被认为是同一物种。但是这个模型中的代理人不进行有性繁殖，所以这个定义不适用。

在无性繁殖的生物体中，如细菌，物种的分布并不是那么清晰。一般来说，如果一个种群的基因型形成一个簇，即种群内的遗传差异比种群间的差异小，则该种群被认为是一个种群。

在我们为新物种建立模型之前，我们需要有能力识别景观中的群落，这意味着我们需要一个距离的确定

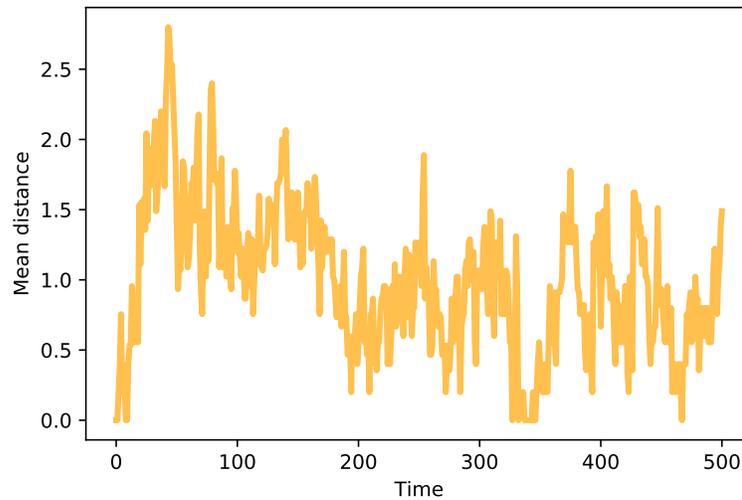


Figure 11.5: Mean distance between agents over time.

locations. Since locations are represented with arrays of bits, we’ll define distance as the number of bits that differ between locations. `FitnessLandscape` provides a `distance` method:

```
# class FitnessLandscape

def distance(self, loc1, loc2):
    return np.sum(np.logical_xor(loc1, loc2))
```

The `logical_xor` function computes “exclusive OR”, which is `True` for bits that differ, and `False` for the bits that are the same.

To quantify the dispersion of a population, we can compute the mean of the distances between pairs of agents. In the notebook for this chapter, you’ll see the `MeanDistance` instrument, which computes this metric after each time step.

Figure 11.5 shows mean distance between agents over time. Because we start with identical mutants, the initial distances are 0. As mutations occur, mean distance increases, reaching a maximum while the population migrates across the landscape.

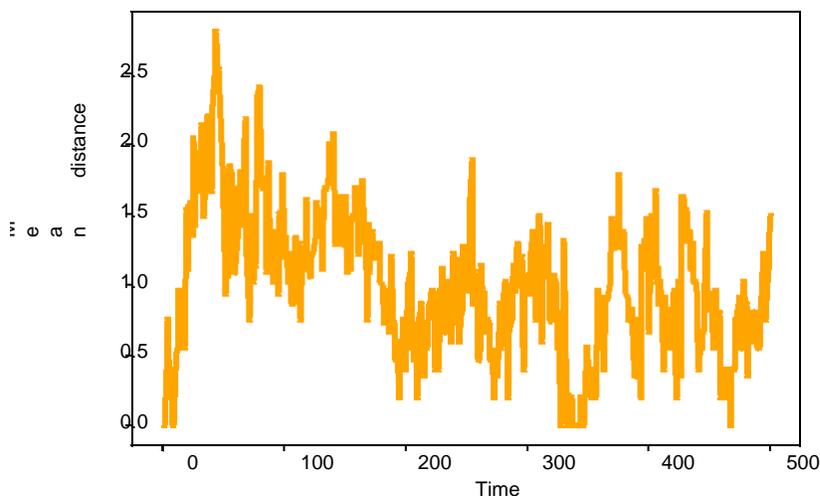


图 11.5: 随着时间推移代理之间的平均距离。

位置。由于位置是用位数组表示的，所以我们将距离定义为位置之间的位数。FitnessLandscape 提供了一种距离方法：

```
# 班级合适/景观
```

```
Def distance (self, loc1, loc2):
```

```
    返回 np.sum (np.logical _ xor (loc1, loc2))
```

逻辑 `_xor` 函数计算 `exclusive OR`”，对于 `different` 的位为 `True`，对于相同的位为 `False`。

为了量化种群的离散度，我们可以计算成对代理之间距离的平均值。在本章的笔记本中，您将看到 `MeanDistance` 工具，它在每个时间步骤之后计算这个度量。

图 11.5 显示了随着时间推移代理之间的平均距离。因为我们从相同的突变体开始，初始距离是 0。随着突变的发生，平均距离增加，达到一个最大值，而种群迁移的景观。

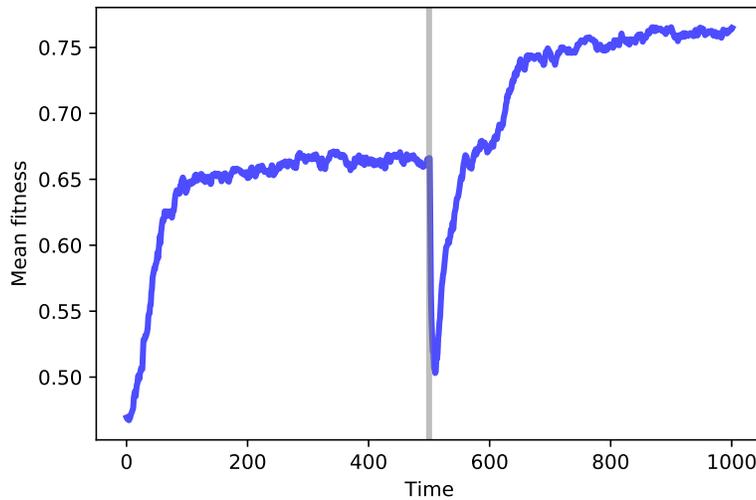


Figure 11.6: Mean fitness over time. After 500 time steps, we change the fitness landscape.

Once the agents discover the optimal location, mean distance decreases until the population reaches an equilibrium where increasing distance due to mutation is balanced by decreasing distance as agents far from the optimal location disappear. In these simulations, the mean distance in equilibrium is near 1; that is, most agents are only one mutation away from optimal.

Now we are ready to look for new species. To model a simple kind of speciation, suppose a population evolves in an unchanging environment until it reaches steady state (like some species we find in nature that seem to have changed very little over long periods of time).

Now suppose we either change the environment or transport the population to a new environment. Some features that increased fitness in the old environment might decrease it in the new environment, and vice versa.

We can model these scenarios by running a simulation until the population reaches steady state, then changing the fitness landscape, and then resuming the simulation until the population reaches steady state again.

Figure 11.6 shows results from a simulation like that. We start with 100 identical mutants at a random location, and run the simulation for 500 time

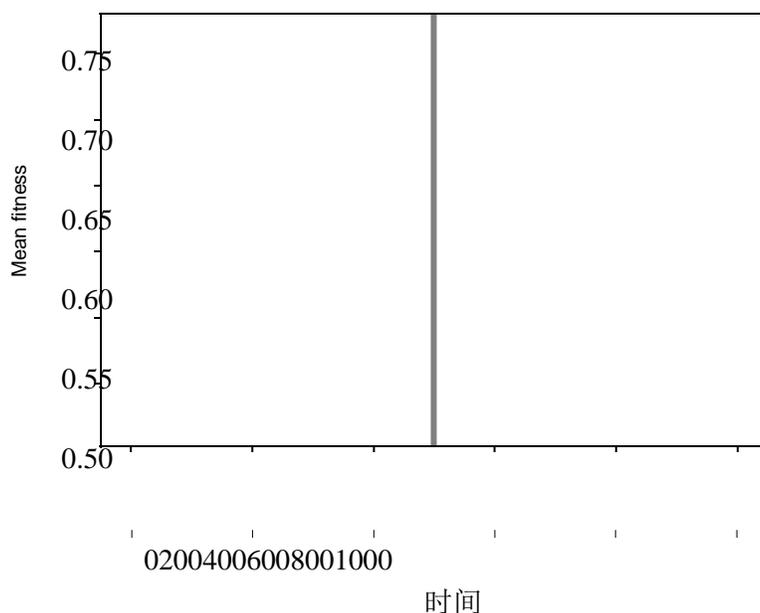


图 11.6: 随时间变化的平均值。经过 500 个时间步骤后，我们改变了前景。

一旦代理人发现最佳位置，平均距离减少，直到种群达到一个平衡，其中增加的距离由于突变是平衡的减少距离代理人远离最佳位置消失。在这些模拟中，平衡点的平均距离接近 1，也就是说，大多数个体离最优只差一个变异。

现在我们准备寻找新的物种。为了建立一个简单的物种形成模型，假设一个种群在一个不变的环境中进化，直到它达到稳定状态(就像我们在自然界中发现的一些物种，在很长时间内似乎变化很小)。

现在假设我们要么改变环境，要么把人口转移到一个新的环境。在旧环境中增加的一些特性可能会在新环境中降低它，反之亦然。

我们可以通过运行一个模拟直到种群达到稳定状态，然后改变前景，然后恢复模拟直到种群再次达到稳定状态来模拟这些情景。

图 11.6 显示了类似的模拟结果。我们从随机位置的 100 个相同的突变体开始，运行 500 次模拟

steps. At that point, many agents are at the optimal location, which has fitness near 0.65 in this example. The genotypes of the agents form a cluster, with the mean distance between agents near 1.

After 500 steps, we run `FitnessLandscape.set_values`, which changes the fitness landscape; then we resume the simulation. Mean fitness is lower, as we expect because the optimal location in the old landscape is no better than a random location in the new landscape.

After the change, mean fitness increases again as the population migrates across the new landscape, eventually finding the new optimum, which has fitness near 0.75 (which happens to be higher in this example, but needn't be).

Once the population reaches steady state, it forms a new cluster, with mean distance between agents near 1 again.

Now if we compute the distance between the agents' locations before and after the change, they differ by more than 6, on average. The distances between clusters are much bigger than the distances between agents in each cluster, so we can interpret these clusters as distinct species.

## 11.10 Summary

We have seen that mutation, along with differential survival and reproduction, is sufficient to cause increasing fitness, increasing diversity, and a simple form of speciation. This model is not meant to be realistic; evolution in natural systems is much more complicated than this. Rather, it is meant to be a “sufficiency theorem”; that is, a demonstration that the features of the model are sufficient to produce the behavior we are trying to explain (see <http://thinkcomplex.com/suff>).

Logically, this “theorem” doesn't prove that evolution in nature is caused by these mechanisms alone. But since these mechanisms do appear, in many forms, in biological systems, it is reasonable to think that they at least contribute to natural evolution.

Likewise, the model does not prove that these mechanisms always cause evolution. But the results we see here turn out to be robust: in almost any

步骤。在这一点上，许多代理都处于最佳位置，在这个例子中，这个位置接近 0.65。药剂的基因型形成一个簇，药剂之间的平均距离接近 1。

500 步之后，我们运行 `fitnesslandscape.set_value`，它改变了前景；然后我们继续模拟。正如我们所期望的那样，平均值较低，因为在旧景观中的最佳位置并不比在新景观中的随机位置好。

在变化之后，平均值随着人口在新地形中的迁移而再次增加，最终得到新的最佳值，这个最佳值接近 0.75(在这个例子中，这个值碰巧更高，但不需要)。

一旦种群达到稳定状态，它就形成一个新的集群，代理之间的平均距离再次接近 1。

现在如果我们计算代理人位置变化前后的距离，他们的平均值超过 6。簇之间的距离比每个簇中代理之间的距离大得多，因此我们可以将这些簇解释为不同的物种。

### 11.10 摘要

我们已经看到，基因突变，伴随着微生物的存活和繁殖，会导致物种数量的增加，物种多样性的增加，以及一种简单的物种形成。这个模型并不现实；自然系统的进化要比这复杂得多。更确切地说，它是一个超 [thinkcomplex.com/suff](http://thinkcomplex.com/suff) 定理”，也就是说，一个证明模型的特征是绝对可以产生我们试图解释的行为的证明。

逻辑上，这个定理”并不能证明自然界的进化仅仅是由这些机制引起的。但是，由于这些机制确实以多种形式出现在生物系统中，因此有理由认为它们至少对自然进化有贡献。

同样，该模型并不能证明这些机制总是导致进化。但是，我们在这里看到的结果证明是健全的：在几乎任何

model that includes these features — imperfect replicators, variability, and differential reproduction — evolution happens.

I hope this observation helps to demystify evolution. When we look at natural systems, evolution seems complicated. And because we primarily see the results of evolution, with only glimpses of the process, it can be hard to imagine and hard to believe.

But in simulation, we see the whole process, not just the results. And by including the minimal set of features to produce evolution — temporarily ignoring the vast complexity of biological life — we can see evolution as the surprisingly simple, inevitable idea that it is.

## 11.11 Exercises

The code for this chapter is in the Jupyter notebook `chap11.ipynb` in the repository for this book. Open the notebook, read the code, and run the cells. You can use the notebook to work on the following exercises. My solutions are in `chap11soln.ipynb`.

**Exercise 11.1** The notebook shows the effects of differential reproductions and survival separately. What if you have both? Write a class called `SimWithBoth` that uses the version of `choose_dead` from `SimWithDiffSurvival` and the version of `choose_replacements` from `SimWithDiffReproduction`. Does mean fitness increase more quickly?

As a Python challenge, can you write this class without copying code?

**Exercise 11.2** When we change the landscape as in Section 11.9, the number of occupied locations and the mean distance usually increase, but the effect is not always big enough to be obvious. Try out some different random seeds to see how general the effect is.

模型包括这些特征 | 不完美的复制因子，可变性，和双向繁殖 | 进化发生。

我希望这个观察结果能够帮助揭开进化论的神秘面纱。当我们观察自然系统时，进化看起来很复杂。因为我们主要看到的是进化的结果，而仅仅是进化过程的一瞥，这很难想象，也很难相信。

但是在模拟中，我们看到的是整个过程，而不仅仅是结果。通过加入最小的一系列特征来产生进化 | 暂时忽略了生物生命的巨大复杂性 | 我们可以看到进化是一个令人惊讶的简单，不可避免的想法。

### 11.11 练习

本章的代码在本书资料库中的 Jupyter 笔记本 `chap11.ipynb` 中。打开笔记本，阅读代码，并运行单元格。你可以用这个笔记本做以下练习。我的解决方案在第 11 章。

练习 11.1 这本笔记本分别展示了微生物的复制和存活情况。如果你两者都有呢？编写一个名为 `SimWithBoth` 的类，它使用 `SimWithDiffSurvival` 中的 `choose dead` 版本和 `SimWithDiffReproduction` 中的 `choose replacements` 版本。是否意味着增长更快？

作为一个 Python 挑战，你能不复制代码就编写这个类吗？

练习 11.2 当我们像 11.9 节那样改变地形时，占据地点的数量和平均距离通常会增加，但是这些影响并不总是大到足以显而易见。尝试一些不同的随机种子，看看这个方面有多普遍。

# Chapter 12

## Evolution of cooperation

In this final chapter, I take on two questions, one from biology and one from philosophy:

- In biology, the “problem of altruism” is the apparent conflict between natural selection, which suggests that animals live in a state of constant competition, and altruism, which is the tendency of many animals to help other animals, even to their own detriment. See <http://thinkcomplex.com/altruism>.
- In moral philosophy, the question of human nature asks whether humans are fundamentally good, or evil, or blank states shaped by their environment. See <http://thinkcomplex.com/nature>.

The tools I use to address these questions are agent-based simulation (again) and game theory, which is a set of abstract models meant to describe ways agents interact. Specifically, the game we will consider is the Prisoner’s Dilemma.

The code for this chapter is in `chap12.ipynb`, which is a Jupyter notebook in the repository for this book. For more information about working with this code, see Section 0.3.

## 第十二章

### 合作的演变

在这最后一章，我回答了两个问题，一个来自生物学，另一个来自哲学：

在生物学中，“利他主义问题”是自然选择和利他主义之间的明显冲突，前者表明动物生活在一种持续竞争的状态中，而后者表明许多动物倾向于帮助其他动物，即使对自己有害。参见 <http://thinkcomplex>。

无私奉献。

在道德哲学中，人性的问题在于人本质上是善还是恶，或者是由环境塑造的空白状态。参见 <http://thinkcomplex.com/nature>。

我用来解决这些问题的工具是基于代理的模拟(再次)和博弈论，这是一组抽象的模型，用来描述代理的互动方式。具体来说，我们将考虑的博弈是囚徒困境。

本章的代码在 `chap12.ipynb` 中，这是本书资料库中的一个 Jupyter 笔记本。有关使用此代码的更多信息，请参见 0.3 节。

## 12.1 Prisoner's Dilemma

The Prisoner's Dilemma is a topic in game theory, but it's not the fun kind of game. Instead, it is the kind of game that sheds light on human motivation and behavior. Here is the presentation of the dilemma from Wikipedia (<http://thinkcomplex.com/pd>):

Two members of a criminal gang are arrested and imprisoned. Each prisoner is in solitary confinement with no means of communicating with the other. The prosecutors lack sufficient evidence to convict the pair on the principal charge, but they have enough to convict both on a lesser charge. Simultaneously, the prosecutors offer each prisoner a bargain. Each prisoner is given the opportunity to either: (1) betray the other by testifying that the other committed the crime, or (2) cooperate with the other by remaining silent. The offer is:

- If A and B each betray the other, each of them serves 2 years in prison.
- If A betrays B but B remains silent, A will be set free and B will serve 3 years in prison (and vice versa).
- If A and B both remain silent, both of them will only serve 1 year in prison (on the lesser charge).

Obviously, this scenario is contrived, but it is meant to represent a variety of interactions where agents have to choose whether to “cooperate” with each other or “defect”, and where the reward (or punishment) for each agent depends on what the other chooses.

With this set of punishments, it is tempting to say that the players should cooperate, that is, that both should remain silent. But neither agent knows what the other will do, so each has to consider two possible outcomes. First, looking at it from A's point of view:

- If B remains silent, A is better off defecting; she would go free rather than serve 1 year.
- If B defects, A is still better off defecting; she would serve only 2 years rather than 3.

### 12.1 囚徒困境

囚徒困境是博弈论中的一个话题，但它并不是一种有趣的博弈。相反，这是一种揭示人类动机和行为的博弈。下面是来自维基百科([http:// thinkcomplex.com/pd](http://thinkcomplex.com/pd))的关于这个困境的介绍:

一个犯罪团伙的两名成员被逮捕和监禁。每个囚犯都被单独关押，没有与其他囚犯联系的手段。检察官缺乏确凿的证据来定罪两人的主要指控，但他们有足够的证据定罪两个较轻的指控。与此同时，检察官与每个犯人达成了协议。每个囚犯都有机会:

(1) 通过证明对方实施了犯罪而出卖对方，或者(2)通过保持沉默而与对方合作。答案是:

如果 a 和 b 互相出卖对方，他们每个人都要服刑两年。

如果 a 背叛了 b，而 b 保持沉默，a 将被释放，b 将被释放  
会被判入狱三年(反之亦然)。

如果 a 和 b 都保持沉默，他们都只会在监狱里呆一年(以较轻的罪名)。

显然，这个场景是人为设计的，但它意味着代表各种各样的交互作用，在这些交互作用中，代理人必须选择是“相互合作还是背叛”，每个代理人的奖励(或惩罚)取决于对方的选择。

有了这一系列的惩罚，很容易让人联想到球员们应该合作，也就是说，双方都应该保持沉默。但是两个代理都不知道对方会做什么，所以每个代理都必须考虑两种可能的结果。首先，从 a 的角度来看:

如果 b 保持沉默，那么 a 最好还是选择背叛，她会选择自由  
也不愿服刑一年。

如果 b 犯了错误，a 还是选择叛逃，她只会服刑 2 年而不是 3 年。

No matter what B does, A is better off defecting. And because the game is symmetric, this analysis is the same from B's point of view: no matter what A does, B is better off defecting.

In the simplest version of this game, we assume that A and B have no other considerations to take into account. They can't communicate with each other, so they can't negotiate, make promises, or threaten each other. And they consider only the immediate goal of minimizing their sentences; they don't take into account any other factors.

Under those assumptions, the rational choice for both agents is to defect. That might be a good thing, at least for purposes of criminal justice. But for the prisoners, it is frustrating because there is, apparently, nothing they can do to achieve the outcome they both want. And this model applies to other scenarios in real life where cooperation would be better for the greater good as well as for the players.

Studying these scenarios, and ways to escape from the dilemma, is the focus of people who study game theory, but it is not the focus of this chapter. We are headed in a different direction.

## 12.2 The problem of nice

Since the Prisoner's Dilemma was first discussed in the 1950s, it has been a popular topic of study in social psychology. Based on the analysis in the previous section, we can say what a perfectly rational agent *should* do; it is harder to predict what real people actually do. Fortunately, the experiment has been done<sup>1</sup>.

If we assume that people are smart enough to do the analysis (or understand it when explained), and that they generally act in their own interest, we would expect them to defect pretty much all the time. But they don't. In most

---

<sup>1</sup>Here's a recent report with references to previous experiments: Barreda-Tarrazona, Jaramillo-Gutiérrez, Pavan, and Sabater-Grande, "Individual Characteristics vs. Experience: An Experimental Study on Cooperation in Prisoner's Dilemma", *Frontiers in Psychology*, 2017; 8: 596. <http://thinkcomplex.com/pdexp>.

无论 b 做什么, a 都比 b 叛逃好。因为博弈是对称的, 所以这个分析从 b 的角度来看是一样的: 无论 a 做什么, b 都是更好的叛逃者。

在这个博弈的最简单版本中, 我们假设 a 和 b 没有其他考虑因素要考虑。他们不能相互交流, 所以他们不能谈判, 不能做出承诺, 也不能相互威胁。而且他们只考虑最小化他们的句子这个直接目标; 他们没有考虑任何其他因素。

在这些假设下, 双方的理性选择就是背叛。这可能是一件好事, 至少对于刑事司法来说是这样。但对于囚犯来说, 这是令人沮丧的, 因为显然, 他们无法做任何事情来达到双方都想要的结果。这个模型也适用于现实生活中的其他场景, 在这些场景中, 合作对于玩家来说更好, 对于玩家来说也更好。

研究这些情景, 以及如何摆脱这种困境, 是博弈论研究者关注的焦点, 但这并不是本章的重点。我们正朝着不同的方向前进。

## 12.2 友善的问题

囚徒困境自 20 世纪 50 年代首次被提出以来, 一直是社会心理学研究的热门话题。基于上一节的分析, 我们可以说一个完全理性的主体应该做什么, 但是很难预测真实的人到底会做什么。幸运的是, 实验已经完成了。

如果我们假设人们足够聪明能够进行分析(或者在解释时能够理解), 并且他们的行为通常符合他们自己的利益, 那么我们会认为他们几乎一直在变节。但事实并非如此。在大多数情况下

---

1 这里有一份最近的报告, 其中提到了以前的实验: Barreda-Tarrazona, Jaramillo-Gutierrez, Pavan, 和 Sabater-Grande, 个人特征 vs. 经验: 关于囚徒困境中合作的实验研究, 《心理学前沿》, 2017; 8:596。Http://thinkcomplex.com/pdexp.

experiments, subjects cooperate much more than the rational agent model predicts<sup>2</sup>.

The most obvious explanation of this result is that people are not rational agents, which should not be a surprise to anyone. But why not? Is it because they are not smart enough to understand the scenario or because they are knowingly acting contrary to their own interest?

Based on experimental results, it seems that at least part of the explanation is plain altruism: many people are willing to incur a cost to themselves in order to benefit another person. Now, before you nominate that conclusion for publication in the *Journal of Obvious Results*, let's keep asking why:

- Why do people help other people, even at a cost to themselves? At least part of the reason is that they want to; it makes them feel good about themselves and the world.
- And why does being nice make people feel good? It might be tempting to say that they were raised right, or more generally trained by society to want to do good things. But there is little doubt that some part of altruism is innate; a proclivity for altruism is the result of normal brain development.
- Well, why is that? The innate parts of brain development, and the personal characteristics that follow, are the result of genetic information. Of course, the relationship between genes and altruism is complicated; there are probably many genes that interact with each other and with environmental factors to cause people to be more or less altruistic in different circumstances. Nevertheless, there are almost certainly genes that tend to make people altruistic.
- Finally, why is that? If, under natural selection, animals are in constant competition with each other to survive and reproduce, it seems obvious that altruism would be counterproductive. In a population where some people help others, even to their own detriment, and others are purely selfish, it seems like the selfish ones would benefit, the altruistic ones would suffer, and the genes for altruism would be driven to extinction.

---

<sup>2</sup>For an excellent video summarizing what we have discussed so far, see <http://thinkcomplex.com/pdvid1>.

实验结果表明，受试者的合作程度远远高于理性代理模型预测的 2。

对这一结果最明显的解释是，人不是理性的行为者，这对任何人来说都不应该感到惊讶。但是为什么不呢？是因为他们不够聪明，不能理解这种情况，还是因为他们明知故犯地违背了自己的利益？

根据实验结果，似乎至少部分解释是纯粹的利他主义：许多人愿意为了帮助他人而付出代价。现在，在你提名这个结论发表在《显而易见的结果》杂志上之前，让我们继续问为什么：

为什么人们要帮助别人，即使要付出自己的代价？至少部分原因是他们想这么做，这让他们感觉良好

他们自己和整个世界。

为什么善良会让人感觉良好？人们可能会说，他们是被正确地教育成长起来的，或者更普遍地说，他们是被社会培养成想要做好事的人。但毫无疑问，利他主义的某些部分是与生俱来的；利他主义的癖性是正常大脑的结果

发展。

为什么？大脑发育的先天部分，以及随之而来的个人特征，都是遗传信息的结果。当然，基因和利他主义之间的关系是复杂的；可能有许多基因相互作用，并与环境因素相互作用，导致人们在不同的环境下或多或少地表现出利他主义。然而，几乎可以肯定存在基因

会让人变得无私。

最后，为什么会这样？如果在自然选择下，动物们为了生存和繁殖而不断地相互竞争，那么利他主义显然会适得其反。在这样一个群体中，有些人帮助别人，甚至有损于他们自己，而其他只是纯粹的自然现象，看起来好像自然现象是有利的，利他主义的人会有利，而利他主义的基因会被推向灭绝。

---

<sup>2</sup> 如果你想看到一个很棒的视频来总结我们到目前为止所讨论的内容，请看 [http:// thinkcomplex.com/pdvid1](http://thinkcomplex.com/pdvid1)。

This apparent contradiction is the “problem of altruism”: *why haven't the genes for altruism died out?*

Among biologists, there are many possible explanations, including reciprocal altruism, sexual selection, kin selection, and group selection. Among non-scientists, there are even more explanations. I leave it to you to explore the alternatives; for now I want to focus on just one explanation, arguably the simplest one: maybe altruism is adaptive. In other words, maybe genes for altruism make people more likely to survive and reproduce.

It turns out that the Prisoner's Dilemma, which raises the problem of altruism, might also help resolve it.

## 12.3 Prisoner's dilemma tournaments

In the late 1970s Robert Axelrod, a political scientist at the University of Michigan, organized a tournament to compare strategies for playing Prisoner's Dilemma (PD).

He invited participants to submit strategies in the form of computer programs, then played the programs against each other and kept score. Specifically, they played the iterated version of PD, in which the agents play multiple rounds against the same opponent, so their decisions can be based on history.

In Axelrod's tournaments, a simple strategy that did surprisingly well was called “tit for tat”, or TFT. TFT always cooperates during the first round of an iterated match; after that, it copies whatever the opponent did during the previous round. If the opponent keeps cooperating, TFT keeps cooperating. If the opponent defects at any point, TFT defects in the next round. But if the opponent goes back to cooperating, so does TFT.

For more information about these tournaments, and an explanation of why TFT does so well, see this video: <http://thinkcomplex.com/pdvid2>.

Looking at the strategies that did well in these tournaments, Alexrod identified the characteristics they tended to share:

- Nice: The strategies that do well cooperate during the first round, and generally cooperate as often as they defect in subsequent rounds.

这个明显的矛盾就是利他主义的问题”：为什么利他主义的基因没有消亡？

在生物学家中，有许多可能的解释，包括互利主义、性选择、亲缘选择和群体选择。在非科学家中，有更多的解释。我把探索替代品的任务留给你们；现在我只想关注一个解释，可以说是最简单的一个：也许利他主义是适应性的。换句话说，也许利他主义的基因使人们更有可能生存和繁殖。

事实证明，提出了利他主义问题的囚徒困境也可能有助于解决这个问题。

### 12.3 囚徒困境锦标赛

20 世纪 70 年代末，密歇根大学的政治学家 Robert Axelrod 组织了一场比赛，比较玩囚徒困境的策略。

他邀请参与者以计算机程序的形式提交策略，然后让程序互相对抗并记分。特别地，他们使用迭代版本的 PD，在这个版本中，特工们对同一个对手进行多轮比赛，因此他们的决定可以基于历史。

在阿克塞尔罗德的锦标赛中，有一个简单的策略取得了意想不到的好成绩，叫做“以牙还牙”，或 TFT。TFT 总是在重复比赛的第一回合中配合；在那之后，它会复制对手在前一回合中的任何动作。如果对手继续合作，TFT 也会继续合作。如果对手在任何一点出现缺陷，TFT 在下一轮出现缺陷。但是如果对手回到合作，TFT 也会回来。

关于这些比赛的更多信息，以及为什么 TFT 做得这么好的解释，请看这个视频：<http://thinkcomplex.com/pdvid2>。

看看在这些锦标赛中表现出色的策略，Axelrod 发现了他们的共同特点：

好的策略：在第一轮中表现良好的策略合作，并且通常在随后的一轮中缺陷的时候也会经常合作。

- Retaliating: Strategies that cooperate all the time did not do as well as strategies that retaliate if the opponent defects.
- Forgiving: But strategies that were too vindictive tended to punish themselves as well as their opponents.
- Non-envious: Some of the most successful strategies seldom outscore their opponents; they are successful because they do *well enough* against a wide variety of opponents.

TFT has all of these properties.

Axelrod's tournaments offer a partial, possible answer to the problem of altruism: maybe the genes for altruism are prevalent because they are adaptive. To the degree that many social interactions can be modeled as variations on the Prisoner's Dilemma, a brain that is wired to be nice, tempered by a balance of retaliation and forgiveness, will tend to do well in a wide variety of circumstances.

But the strategies in Axelrod's tournaments were designed by people; they didn't evolve. We need to consider whether it is credible that genes for niceness, retribution, and forgiveness could appear by mutation, successfully invade a population of other strategies, and resist being invaded by subsequent mutations.

## 12.4 Simulating evolution of cooperation

*Evolution of Cooperation* is the title of the first book where Axelrod presented results from Prisoner's Dilemma tournaments and discussed the implications for the problem of altruism. Since then, he and other researchers have explored the evolutionary dynamics of PD tournaments, that is, how the distribution of strategies changes over time in a population of PD contestants. In the rest of this chapter, I run a version of those experiments and present the results.

First, we'll need a way to encode a PD strategy as a genotype. For this experiment, I consider strategies where the agent's choice in each round depends only on the opponent's choice in the previous two rounds. I represent a strategy using a dictionary that maps from the opponent's previous two choices to the agent's next choice.

报复: 一直合作的策略不如对手出现问题时的报复策略。

宽恕: 但是过于报复的策略往往会惩罚他们-  
自我和他们的对手。

不嫉妒: 一些最成功的策略很少能打败他们的对手, 他们之所以成功是因为他们在对付各种各样的对手时表现得足够好。

TFT 具有所有这些特性。

阿克塞尔罗德的锦标赛是对利他主义问题的一个可能的部分答案: 也许利他主义的基因之所以普遍存在, 是因为它们具有适应性。在某种程度上, 许多社会互动可以被模拟为囚徒困境的变体, 一个与善良相连的大脑, 被报复和宽恕的平衡所调和, 将倾向于在各种各样的情况下表现良好。

但是阿克塞尔罗德锦标赛中的策略是由人们设计的, 他们没有进化。我们需要考虑的是, 美好、报应和宽恕的基因可以通过突变出现, 成功地侵入其他策略的种群, 并抵御随后的突变的侵入, 这种说法是否可信。

#### 12.4 模拟合作的演变

合作的进化》是第一本书的标题, 其中阿克塞尔罗德介绍了囚徒困境联赛的结果, 并讨论了利他主义问题的含义。从那时起, 他和其他研究人员开始探索 PD 锦标赛的进化动力学, 也就是说, 策略的分布是如何随着时间的推移而改变的。在本章的其余部分, 我将运行这些实验的一个版本并展示实验结果。

首先, 我们需要一种方法将 PD 策略作为基因型进行编码。对于这个实验, 我考虑的策略是, 经纪人在每个回合中的选择仅取决于对手在前两个回合中的选择。我使用一个字典来表示一个策略, 该字典将对手的前两个选择映射到代理的下一个选择。

Here is the class definition for these agents:

```
class Agent:

    keys = [(None, None),
            (None, 'C'),
            (None, 'D'),
            ('C', 'C'),
            ('C', 'D'),
            ('D', 'C'),
            ('D', 'D')]

    def __init__(self, values, fitness=np.nan):
        self.values = values
        self.responses = dict(zip(self.keys, values))
        self.fitness = fitness
```

`keys` is the sequence of keys in each agent's dictionary, where the tuple ('C', 'C') means that the opponent cooperated in the previous two rounds; (None, 'C') means that only one round has been played and the opponent cooperated; and (None, None) means that no rounds have been played.

In the `__init__` method, `values` is a sequence of choices, either 'C' or 'D', that correspond to `keys`. So if the first element of `values` is 'C', that means that this agent will cooperate in the first round. If the last element of `values` is 'D', this agent will defect if the opponent defected in the previous two rounds.

In this implementation, the genotype of an agent who always defects is 'DDDDDD'; the genotype of an agent who always cooperates is 'CCCCCC', and the genotype for TFT is 'CCDCDCD'.

The `Agent` class provides `copy`, which makes another agent with the same genotype, but with some probability of mutation:

```
def copy(self, probab_mutate=0.05):
    if np.random.random() > probab_mutate:
        values = self.values
    else:
        values = self.mutate()
    return Agent(values, self.fitness)
```

以下是这些代理人的等级划分:

类别代理:

```
(没有, 没有),
    (无, c),
    (无, d'),
    ( c, c),
    ( c, d),
    ( d, c) , ('
d' , ' d')
```

返回文章页面自我价值, 健身 = np.nan):

```
价值 = 价值
Self.responses = dict (zip (self.keys, values))
健康 = 健康
```

Key 是每个代理词典中键的顺序, 其中 tuple (' c, ' c')表示对方在前两轮中合作; (None, ' c')表示只玩了一轮, 对方合作; (None, None)表示没有玩过回合。

在 `_init _ method` 中, 值是与键对应的选项序列, 可以是 ' c'或' d'。因此, 如果第一个元素的值是 ' c', 这意味着'这个代理将在第一轮合作。如果值的最后一个元素是 d', 则如果对手在前两轮中叛逃, 该代理将会叛逃。

在此实施过程中, 始终缺陷病原体的基因型为“ ddddd”, 始终合作病原体的基因型为“ ccccccc”, TFT 的基因型为“ CCDCDCD”。

代理类提供复制, 这使得另一个代理具有相同的基因型, 但具有一定的变异概率:

```
Def copy (self, prob _ mutate = 0.05) :
    如果 np.random.random () > prob _ mutate:
        价值 = 自己
    其他:
        Value = self.mutate ()
    返回代理(值、自身适应性)
```

Mutation works by choosing a random value in the genotype and flipping from 'C' to 'D', or vice versa:

```
def mutate(self):
    values = list(self.values)
    index = np.random.choice(len(values))
    values[index] = 'C' if values[index] == 'D' else 'D'
    return values
```

Now that we have agents, we need a tournament.

## 12.5 The Tournament

The `Tournament` class encapsulates the details of the PD competition:

```
payoffs = {('C', 'C'): (3, 3),
           ('C', 'D'): (0, 5),
           ('D', 'C'): (5, 0),
           ('D', 'D'): (1, 1)}

num_rounds = 6

def play(self, agent1, agent2):
    agent1.reset()
    agent2.reset()

    for i in range(self.num_rounds):
        resp1 = agent1.respond(agent2)
        resp2 = agent2.respond(agent1)

        pay1, pay2 = self.payoffs[resp1, resp2]

        agent1.append(resp1, pay1)
        agent2.append(resp2, pay2)

    return agent1.score, agent2.score
```

`payoffs` is a dictionary that maps from the agents' choices to their rewards. For example, if both agents cooperate, they each get 3 points. If one defects

基因突变的工作原理是在基因型中选择一个随机值，然后从 C'变成 d'，反之亦然：

```

Def mutate (self) :
    Values = list (self.values)
    Index = np.random.choice (len (values))
    值[索引] = ' c'如果值[索引] == ' d' else' d'
    返回值

```

现在我们有经纪人了，我们需要一个锦标赛。

## 12.5 锦标赛

Tournament 类封装了 PD 竞赛的细节：

```

= {(c, c) : (3,3),
    ( C, d) : (0,5),
    ( d, c) : (5,0), (' d' , '
    d') : (1,1)}

```

6

```

(self, agent 1, agent 2) :
    探员 1.reset ()
    2. reset ()

```

```

对于 i in range (self.num rounds) :
    呼吸 1 = agent1. respond (agent2)
    呼吸 2 = agent2.respond (agent1)

    收益 1, 收益 2 = 自己, 收益[响应 1, 响应 2]

```

```

1.append (resp1, pay1)
2.append (resp2, pay2)

```

and the other cooperates, the defector gets 5 and the cooperator gets 0. If they both defect, each gets 1. These are the payoffs Axelrod used in his tournaments.

The `play` method runs several rounds of the PD game. It uses the following methods from the `Agent` class:

- `reset`: Initializes the agents before the first round, resetting their scores and the history of their responses.
- `respond`: Asks each agent for their response, given the opponent's previous responses.
- `append`: Updates each agent by storing the choices and adding up the scores from successive rounds.

After the given number of rounds, `play` returns the total score for each agent. I chose `num_rounds=6` so that each element of the genotype is accessed with roughly the same frequency. The first element is only accessed during the first round, or one sixth of the time. The next two elements are only accessed during the second round, or one twelfth each. The last four elements are accessed four of six times, or one sixth each, on average.

`Tournament` provides a second method, `melee`, that determines which agents compete against each other:

另一方合作，叛逃者得到 5 分，合作者得到 0 分。如果他们都有缺陷，每人得到 1。这些是阿克塞尔罗德在比赛中使用的。

`Play` 方法在 PD 游戏中运行了几轮，它使用 `Agent` 类中的以下方法：

重置: 在第一轮之前初始化代理，重置他们的分数以及他们的反应历史。

回答: 询问每个代理人的反应，给出对手以前的反应。

Append: 通过存储选项和累加连续回合的得分来更新每个代理。

在给定的回合数之后，`play` 返回每个代理的总分。我选择 `num_rounds = 6`，这样基因型的每个元素被访问的频率大致相同。第一个元素只能在第一轮或六分之一的时间内访问。接下来的两个元素只能在第二轮中访问，或者说每个元素只能访问十二分之一。最后四个元素被访问了六次中的四次，平均每次访问六分之一。

锦标赛提供了第二种方法---- 近战，用来决定哪些代理互相竞争：

```
def melee(self, agents, randomize=True):
    if randomize:
        agents = np.random.permutation(agents)

    n = len(agents)
    i_row = np.arange(n)
    j_row = (i_row + 1) % n

    totals = np.zeros(n)

    for i, j in zip(i_row, j_row):
        agent1, agent2 = agents[i], agents[j]
        score1, score2 = self.play(agent1, agent2)
        totals[i] += score1
        totals[j] += score2

    for i in i_row:
        agents[i].fitness = totals[i] / self.num_rounds / 2
```

`melee` takes a list of agents and a boolean, `randomize`, that determines whether each agent fights the same neighbors every time, or whether the pairings are randomized.

`i_row` and `j_row` contain the indices of the pairings. `totals` contains the total score of each agent.

Inside the loop, we select two agents, invoke `play`, and update `totals`. At the end, we compute the average number of points each agent got, per round and per opponent, and store the results in the `fitness` attribute of each agent.

## 12.6 The Simulation

The `Simulation` class for this chapter is based on the one in Section 11.4; the only differences are in `__init__` and `step`.

Here's the `__init__` method:

```
返回文章页面防御混战(self, agents, randomize = True):
```

```
    如果随机化:
```

```
        代理 = np.random.permutation (agents)
```

```
    (代理人)
```

```
    I_row = np.arange (n)
```

```
    J_row = (i_row + 1)% n
```

```
    Total = np.zeros (n)
```

```
    For i, j in zip (i_row, j_row):
```

```
        探员 1, 探员 2 = 探员[i], 探员[j]
```

```
        得分 1, 得分 2 = 自己
```

```
        总数[i] += 分数
```

```
        总分[j] += 分数
```

```
    因为我坐在第一排:
```

```
        经纪人, 经纪人, 经纪人, 经纪人, 经纪人, 经纪人
```

混战需要一个代理列表和一个布尔值，随机化，决定每个代理是否每次都对应同一个邻居，或者每对代理是否是随机的。

`I_row` 和 `j_row` 包含配对的索引。总数包含每个代理的总分。

在循环中，我们选择两个代理，调用 `play` 和更新汇总。最后，我们计算每个智能体每轮和每个对手的平均得分数，并将结果存储在每个智能体的适应度属性中。

## 12.6 模拟

本章的模拟类是基于第 11.4 节中的类；唯一的差异是在 `_init_` 和步骤中。

下面是 `init` 方法:

```
class PDSimulation(Simulation):  
  
    def __init__(self, tournament, agents):  
        self.tournament = tournament  
        self.agents = np.asarray(agents)  
        self.instruments = []
```

A `Simulation` object contains a `Tournament` object, a sequence of agents, and a sequence of `Instrument` objects (as in Section 11.6).

Here's the `step` method:

```
def step(self):  
    self.tournament.melee(self.agents)  
    Simulation.step(self)
```

This version of `step` uses `Tournament.melee`, which sets the `fitness` attribute for each agent; then it calls the `step` method from the `Simulation` class, reproduced here:

```
# class Simulation  
  
def step(self):  
    n = len(self.agents)  
    fits = self.get_fitnesses()  
  
    # see who dies  
    index_dead = self.choose_dead(fits)  
    num_dead = len(index_dead)  
  
    # replace the dead with copies of the living  
    replacements = self.choose_replacements(num_dead, fits)  
    self.agents[index_dead] = replacements  
  
    # update any instruments  
    self.update_instruments()
```

`Simulation.step` collects the agents' fitnesses in an array; then it calls `choose_dead` to decide which agents die, and `choose_replacements` to decide which agents reproduce.



My simulation includes differential survival, as in Section 11.7, but not differential reproduction. You can see the details in the notebook for this chapter. As one of the exercises, you will have a chance to explore the effect of differential reproduction.

## 12.7 Results

Suppose we start with a population of three agents: one always cooperates, one always defects, and one plays the TFT strategy. If we run `Tournament.melee` with this population, the cooperator gets 1.5 points per round, the TFT agent gets 1.9, and the defector gets 3.33. This result suggests that “always defect” should quickly become the dominant strategy.

But “always defect” contains the seeds of its own destruction. If nicer strategies are driven to extinction, the defectors have no one to take advantage of. Their fitness drops, and they become vulnerable to invasion by cooperators.

Based on this analysis, it is not easy to predict how the system will behave: will it find a stable equilibrium, or oscillate between various points in the genotype landscape? Let’s run the simulation and find out!

I start with 100 identical agents who always defect, and run the simulation for 5000 steps:

```
tour = Tournament()
agents = make_identical_agents(100, list('DDDDDD'))
sim = PDSimulation(tour, agents)
```

Figure 12.1 shows mean fitness over time (using the `MeanFitness` instrument from Section 11.6). Initially mean fitness is 1, because when defectors face each other, they get only 1 point each per round.

After about 500 time steps, mean fitness increases to nearly 3, which is what cooperators get when they face each other. However, as we suspected, this situation is unstable. Over the next 500 steps, mean fitness drops below 2, climbs back toward 3, and continues to oscillate.

The rest of the simulation is highly variable, but with the exception of one big drop, mean fitness is usually between 2 and 3, with the long-term mean close to 2.5.

我的模拟包括了与第 11.7 节一样的指向性生存，但不包括指向性生殖。你可以在这一章的笔记本上看到细节。作为练习的一部分，你将有机会探索文献复制的方面。

### 12.7 结果

假设我们从三个代理开始：一个总是合作，一个总是缺陷，一个采用 TFT 策略。如果我们和这些人进行竞赛，合作者每轮得 1.5 分，TFT 代理得 1.9 分，叛逃者得 3.33 分。这一结果表明，“总是缺陷”应该很快成为主导策略。

“但总是缺陷”包含着自我毁灭的种子。如果更好的策略被迫消失，那么叛逃者就没有人可以利用了。它们的情绪下降，变得容易受到合作者的入侵。

基于这种分析，不容易预测系统将如何表现：它将找到一个稳定的平衡，或在基因型景观的不同点之间振荡？让我们运行模拟和发现！

我从 100 个总是叛逃的相同代理开始，运行 5000 个步骤的模拟：

```
巡回赛 = 锦标赛()
代理 = make_reconvinced_agents(100, list('ddd'))
模拟(旅游, 代理)
```

图 12.1 显示了随时间变化的平均值(使用 11.6 节中的 `MeanFitness` 仪器)。最初的平均值是 1，因为当背叛者面对面时，他们每轮只能得到 1 分。

经过大约 500 个步骤后，平均线性增加到近 3，这是合作者得到什么，当他们面对对方。然而，正如我们所料，这种情况并不稳定。在接下来的 500 步中，平均值下降到 2 以下，回升到 3，并继续振荡。

其余的模拟是高度可变的，但是除了一个大的下降，平均值通常在 2 到 3 之间，长期平均值接近 2.5。

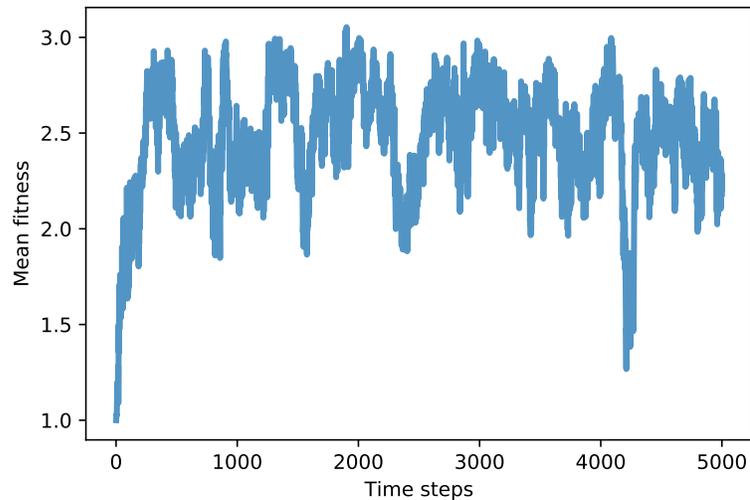


Figure 12.1: Average fitness (points scored per round of Prisoner’s Dilemma).

And that’s not bad! It’s not quite a utopia of cooperation, which would average 3 points per round, but it’s a long way from the dystopia of perpetual defection. And it’s a lot better than what we might expect from the natural selection of self-interested agents.

To get some insight into this level of fitness, let’s look at a few more instruments. `Niceness` measures the fraction of cooperation in the genotypes of the agents after each time step:

```
class Niceness(Instrument):

    def update(self, sim):
        responses = np.array([agent.values
                               for agent in sim.agents])
        metric = np.mean(responses == 'C')
        self.metrics.append(metric)
```

`responses` is an array with one row for each agent and one column for each element of the genome. `metric` is the fraction of elements that are 'C', averaged across agents.

Figure 12.2 (left) shows the results: starting from 0, average niceness increases

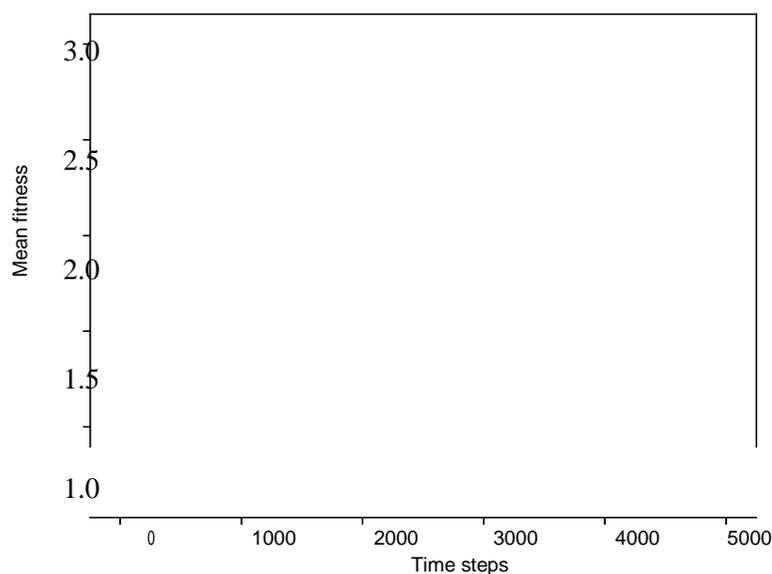


图 12.1: 平均线性(囚徒困境每轮得分)。

这还不错！这不完全是个合作的乌托邦，平均每轮投票得 3 分，但是离永久背叛的反乌托邦还有很长的路要走。这比我们从自利动物的自然选择中所期望的要好得多。

为了深入了解这一层次的特性，让我们看看其他一些工具。在每个时间步骤之后，细度度量了药物基因型中合作的比例：

课程 Niceness (仪器)：

```
Def update (self, sim) :  
    Responses = np.array ([ agent.values  
                           为了西蒙探员])  
    度量 = np.mean (responses = ' c')  
    Self.metrics. append (metric)
```

响应是一个数组，每个代理一行，基因组的每个元素一列。度量是以“c”为单位的元素的分数，跨代理求平均值。

图 12.2(左)显示了结果: 从 0 开始，平均美好度增加

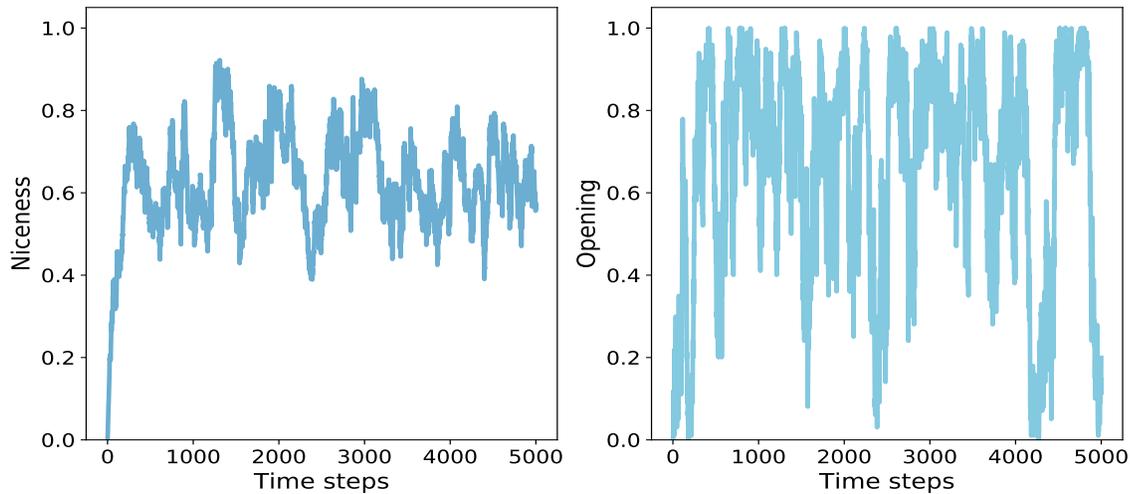


Figure 12.2: Average niceness across all genomes in the population (left), and fraction of population that cooperates in the first round (right).

quickly to 0.75, then oscillates between 0.4 and 0.85, with a long-term mean near 0.65. Again, that’s a lot of niceness!

Looking specifically at the opening move, we can track the fraction of agents that cooperate in the first round. Here’s the instrument:

```
class Opening(Instrument):

    def update(self, sim):
        responses = np.array([agent.values[0]
                               for agent in sim.agents])
        metric = np.mean(responses == 'C')
        self.metrics.append(metric)
```

Figure 12.2 (right) shows the results, which are highly variable. The fraction of agents who cooperate in the first round is often near 1, and occasionally near 0. The long-term average is close to 0.65, similar to overall niceness. These results are consistent with Axelrod’s tournaments; in general, nice strategies do well.

The other characteristics Axelrod identifies in successful strategies are retali-

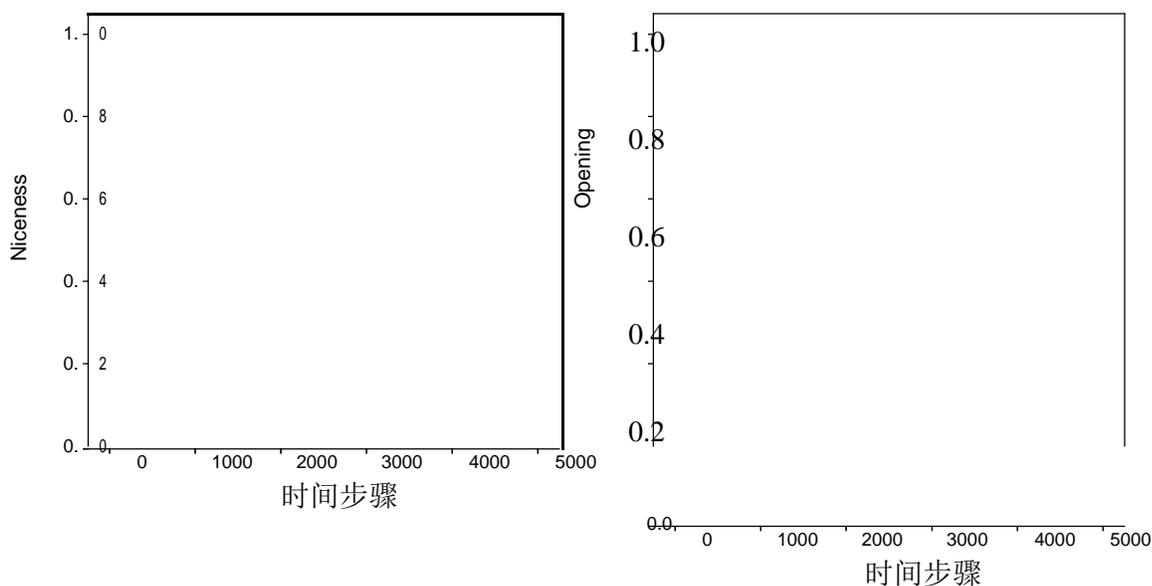


图 12.2: 人口(左)中所有基因组的平均友好度, 以及在第一轮(右)中合作的人口比例。

然后在 0.4 和 0.85 之间振荡, 长期平均值接近 0.65。再说一次, 那是很多美好的事情!

特别地看一下开始的行动, 我们可以追踪第一轮合作的代理商的比例。下面是这个工具:

类别开放(文书):

```

Def update (self, sim):
    Responses = np.array ([ agent.values [0]
                           为了西蒙探员])
    度量 = np.mean (responses = ' c')
    Self.metrics. append (metric)

```

图 12.2(右)显示了结果, 这是高度可变的。在第一轮中合作的代理人的比例通常接近 1, 偶尔也接近 1

0. 长期的平均值接近 0.65, 与整体的美好程度相似。这些结果与阿克塞尔罗德的锦标赛是一致的; 一般来说, 好的策略做得很好。

ation and forgiveness. To measure retaliation, I define this instrument:

```
class Retaliating(Instrument):  
  
    def update(self, sim):  
        after_d = np.array([agent.values[2::2]  
                            for agent in sim.agents])  
        after_c = np.array([agent.values[1::2]  
                            for agent in sim.agents])  
        metric = np.mean(after_d=='D') - np.mean(after_c=='D')  
        self.metrics.append(metric)
```

`Retaliating` compares the number of elements in all genomes where an agent defects after the opponent defects (elements 2, 4, and 6) with the number of places where an agents defects after the opponent cooperates. As you might expect by now, the results vary substantially (you can see the graph in the notebook). On average the difference between these fractions is less than 0.1, so if agents defect 30% of the time after the opponent cooperates, they might defect 40% of the time after a defection.

This result provides weak support for the claim that successful strategies retaliate. But maybe it's not necessary for all agents, or even many, to be retaliatory; if there is at least some tendency toward retaliation in the population as a whole, that might be enough to prevent high-defection strategies from gaining ground.

To measure forgiveness, I define one more instrument to see whether agents might be more likely to cooperate after D-C in the previous two rounds, compared to C-D. In my simulations, there is no evidence for this particular kind of forgiveness. On the other hand, the strategies in these simulations are necessarily forgiving because they consider only the previous two rounds of history. In this context, forgetting is a kind of forgiving.

## 12.8 Conclusions

Axelrod's tournaments suggest a possible resolution to the problem of altruism: maybe being nice, but not *too* nice, is adaptive. But the strategies in the

为了衡量报复，我准备了这个工具：

类别报复(仪器)：

```
Def update (self, sim):  
    在 _d = np.array ([ agent.values [2: : 2]之后  
        为了西蒙探员)  
    在 _c = np.array ([ agent.values [1: : 2]之后  
        为了西蒙探员)  
    Metric = np.mean (after _d == 'd')-np.mean (after _c == 'd')  
    Self.metrics. append (metric)
```

报复将所有基因组中代理人在对手缺陷后缺陷的元素数量(元素 2、4 和 6)与代理人在对手合作后缺陷的位置数量进行比较。正如您现在可能预期的那样，结果差别很大(您可以在笔记本中看到图表)。平均而言，这些分数之间的差异小于 0.1，因此，如果代理人在对手合作后 30% 的时间内叛逃，他们可能在叛逃后 40% 的时间内叛逃。

这一结果为成功的报复战略的说法提供了微弱的支持。但也许并非所有特工，甚至很多特工，都有必要采取报复行动；如果整个民众至少存在一定程度的报复倾向，这可能足以防止高度叛逃策略获得支持。

为了测量宽恕程度，我又多了一个工具来观察在前两轮 D-C 之后，与 C-D 相比，代理人是否更有可能合作。在我的模拟中，没有证据表明这种特殊的宽恕。另一方面，这些模拟中的策略必然是宽容的，因为他们只考虑了前两轮的历史。在这种情况下，遗忘是一种宽恕。

## 12.8 结论

阿克塞尔罗德的锦标赛为利他主义问题提供了一个可能的解决方案：也许友善但不太友善是适应性的。但是这些策略

original tournaments were designed by people, not evolution, and the distribution of strategies did not change over the course of the tournaments.

So that raises a question: strategies like TFT might do well in a fixed population of human-designed strategies, but can they evolve? In other words, can they appear in a population through mutation, compete successfully with their ancestors, and resist invasion by their descendants?

The simulations in this chapter suggest:

- Populations of defectors are vulnerable to invasion by nicer strategies.
- Populations that are too nice are vulnerable to invasion by defectors.
- As a result, the average level of niceness oscillates, but the average amount of niceness is generally high, and the average level of fitness is generally closer to a utopia of cooperation than to a dystopia of defection.
- TFT, which was a successful strategy in Alexrod's tournaments, does not seem to be a specially optimal strategy in an evolving population. In fact, there is probably no stable optimal strategy.
- Some degree of retaliation may be adaptive, but it might not be necessary for all agents to retaliate. If there is enough retaliation in the population as a whole, that might be enough to prevent invasion by defectors<sup>3</sup>.

Obviously, the agents in these simulations are simple, and the Prisoner's Dilemma is a highly abstract model of a limited range of social interactions. Nevertheless, the results in this chapter provide some insight into human nature. Maybe our inclinations toward cooperation, retaliation, and forgiveness are innate, at least in part. These characteristics are a result of how our brains are wired, which is controlled by our genes, at least in part. And maybe our genes build our brains that way because over the history of human evolution, genes for less altruistic brains were less likely to propagate.

Maybe that's why selfish genes build altruistic brains.

---

<sup>3</sup>And that introduces a whole new topic in game theory, the free-rider problem (see <http://thinkcomplex.com/rider>)

最初的锦标赛是由人设计的，而不是进化，并且策略的分布并没有随着锦标赛的进行而改变。

所以这就提出了一个问题：像 TFT 这样的策略可能在人类设计的大量战略中表现良好，但是它们能进化吗？换句话说，他们能否通过突变出现在一个种群中，成功地与他们的祖先竞争，并抵抗他们的后代的入侵？

本章的模拟表明：

叛逃者的人口很容易受到更好的策略的入侵。

过于友善的种群很容易受到叛逃者的入侵。

因此，平均的友好程度会起伏不定，但平均的友好程度通常很高，平均的友好程度通常更接近于合作的乌托邦，而不是反乌托邦感染。

TFT 在 Alexrod 的锦标赛中是一个成功的战略，但在一个不断演变的种群中，它似乎并不是一个特别理想的战略。事实上，可能没有稳定的最优策略。

某种程度的报复可能是适应性的，但可能没有必要让所有特工都去报复。如果整个人口中存在足够的报复行为，这可能足以防止叛逃者的入侵。

显然，这些模拟中的主体都很简单，囚徒困境是一个高度抽象的模型，描述了有限范围内的社会互动。然而，本章的结果提供了一些关于人性的洞察力。也许我们对于合作、报复和宽恕的倾向是与生俱来的，至少在某种程度上是这样。这些特征是我们的大脑是如何连接的结果，这是由我们的基因控制的，至少在一定程度上。也许我们的基因就是这样构建我们的大脑的，因为在人类进化的历史中，利他主义程度较低的大脑基因不太可能传播。

也许这就是为什么 sel sh 基因会产生利他主义的大脑。

---

这在博弈论中引入了一个全新的话题，搭便车问题 ([Http://thinkcomplex.com/rider](http://thinkcomplex.com/rider))

## 12.9 Exercises

The code for this chapter is in the Jupyter notebook `chap12.ipynb` in the repository for this book. Open the notebook, read the code, and run the cells. You can use this notebook to work on the following exercises. My solutions are in `chap12soln.ipynb`.

**Exercise 12.1** The simulations in this chapter depend on conditions and parameters I chose arbitrarily. As an exercise, I encourage you to explore other conditions to see what effect they have on the results. Here are some suggestions:

1. Vary the initial conditions: instead of starting with all defectors, see what happens if you start with all cooperators, all TFT, or random agents.
2. In `Tournament.melee`, I shuffle the agents at the beginning of each time step, so each agent plays against two randomly-chosen agents. What happens if you don't shuffle? In that case, each agent plays against the same neighbors repeatedly. That might make it easier for a minority strategy to invade a majority, by taking advantage of locality.
3. Since each agent only plays against two other agents, the outcome of each round is highly variable: an agent that would do well against most other agents might get unlucky during any given round, or the other way around. What happens if you increase the number of opponents each agent plays against during each round? Or what if an agent's fitness at the end of each step is the average of its current score and its fitness at the end of the previous round?
4. The function I chose for `prob_survival` varies from 0.7 to 0.9, so the least fit agent, with `p=0.7`, lives for 3.33 time steps on average, and the most fit agent lives for 10 time steps. What happens if you make the degree of differential survival more or less "aggressive"?
5. I chose `num_rounds=6` so that each element of the genome has roughly the same impact on the outcome of a match. But that is substantially shorter than what Alexrod used in his tournaments. What happens if you increase `num_rounds`? Note: if you explore the effect of this parameter, you might want to modify `Niceness` to measure the niceness of the last

## 12.9 练习

本章的代码在本书资料库中的 `chapyter` 笔记本 `chap12.ipynb` 中。打开笔记本，阅读代码，并运行单元格。你可以用这个笔记本做以下练习。我的解答在第 12 章。

练习 12.1 本章中的模拟取决于我随意选择的条件和参数。作为练习，我鼓励你探索其他条件，看看它们对结果有什么影响。以下是一些建议：

1. 改变初始条件: 不要从所有的背叛者开始，看看如果你从所有的合作者、所有的 TFT 或者随机的代理人开始会发生什么。
2. 在 `Tournament.melee` 游戏中，我在每个时间步骤的开始处设置代理，因此每个代理对两个随机选择的代理进行游戏。如果你不这么做会发生什么？在这种情况下，每个代理反复对同一个邻居发挥作用。这可能会使少数人的策略更容易侵占多数人，利用地点优势。
3. 由于每个经纪人只和另外两个经纪人打交道，每一轮的结果都有很大的不同：一个经纪人如果在任何一轮中对其他经纪人都表现出色，那么他就有可能在任何一轮中运气不佳，或者恰恰相反。如果你增加每个经纪人在每一轮中对抗的对手的数量会发生什么？或者，如果一个经纪人在每一步结束时的表现是其当前得分和上一轮结束时得分的平均值呢？
4. 我为 `prob` 存活选择的函数从 0.7 到 0.9 不等，因此  $p = 0.7$  的最小 `t` 代理平均存活 3.33 个时间步长，而最大 `t` 代理平均存活 10 个时间步长。如果你让参与生存的程度变得更具攻击性或更弱，会发生什么？”？
5. 我选择 `num rounds = 6`，因此基因组的每个元素对匹配结果的影响大致相同。但是这比阿列克斯罗德在锦标赛中使用的时间要短得多。如果增加 `num_rounds` 会发生什么？注意：如果您研究了参数的影响，您可能希望修改 `Niceness` 来测量最后一个参数的 `Niceness`

4 elements of the genome, which will be under more selective pressure as `num_rounds` increases.

6. My implementation has differential survival but not differential reproduction. What happens if you add differential reproduction?

**Exercise 12.2** In my simulations, the population never converges to a state where a majority share the same, presumably optimal, genotype. There are two possible explanations for this outcome: one is that there is no optimal strategy, because whenever the population is dominated by a majority genotype, that condition creates an opportunity for a minority to invade; the other possibility is that the mutation rate is high enough to maintain a diversity of genotypes.

To distinguish between these explanations, try lowering the mutation rate to see what happens. Alternatively, start with a random population and run without mutation until only one genotype survives. Or run with mutation until the system reaches something like a steady state; then turn off mutation and run until there is only one surviving genotype. What are the characteristics of the genotypes that prevail in these conditions?

**Exercise 12.3** The agents in my experiment are “reactive” in the sense that their choice during each round depends only on what the opponent did during previous rounds. Explore strategies that also take into account the agent’s past choices. These strategies can distinguish an opponent who retaliates from an opponent who defects without provocation.

基因组的 4 个元素，随着数量的增加，这些元素将受到更多的选择压力。

6. 我的实现只有差异生存而没有差异生殖，如果加上差异生殖会怎么样？

练习 12.2 在我的模拟实验中，人口绝不会收敛到大多数人都拥有相同的、可能是最佳的基因型的状态。对于这一结果有两种可能的解释：一种是没有最佳策略，因为每当群体由多数基因型主导时，这种情况就为少数人入侵创造了机会；另一种可能性是突变率高到足以维持基因型的多样性。

为了区分这些解释，试着降低突变率，看看会发生什么。或者，从一个随机的群体开始，不进行突变，直到只有一个基因型存活下来。或者继续进行突变，直到系统达到某种稳定状态；然后转 o 突变，继续进行，直到只有一个存活的基因型。在这些条件下普遍存在的基因型有哪些特征？

练习 12.3 我实验中的代理是反应性的，即他们在每一轮中的选择只取决于对手在前几轮中的所作所为。探索策略，也要考虑到代理人过去的选择。这些策略可以区分对手谁的报复与对手谁的缺陷没有挑衅。

# Appendix A

## Reading list

The following are selected books related to topics in this book. Most are written for a non-technical audience.

- Axelrod, Robert, *Complexity of Cooperation*, Princeton University Press, 1997.
- Axelrod, Robert *The Evolution of Cooperation*, Basic Books, 2006.
- Bak, Per *How Nature Works*, Copernicus (Springer), 1996.
- Barabási, Albert-László, *Linked*, Perseus Books Group, 2002.
- Buchanan, Mark, *Nexus*, W. W. Norton & Company, 2002.
- Dawkins, Richard, *The Selfish Gene*, Oxford University Press, 2016.
- Epstein, Joshua and Axtell, Robert, *Growing Artificial Societies*, Brookings Institution Press & MIT Press, 1996.
- Fisher, Len, *The Perfect Swarm*, Basic Books, 2009.
- Flake, Gary William, *The Computational Beauty of Nature*, MIT Press 2000.
- Goldstein, Rebecca, *Incompleteness*, W. W. Norton & Company, 2005.
- Goodwin, Brian *How the Leopard Changed Its Spots*, Princeton University Press, 2001.

## 附录 a

### 阅读清单

以下是与本书主题相关的精选书籍。大多数是为非技术读者写的。

合作的复杂性》，普林斯顿大学出版社，1997。

合作的进化》，Basic Books，2006。

贝克，《自然是如何运作的》，哥白尼(施普林格)，1996。

珀修斯图书集团，2002年。

威廉·沃德尔·诺顿公司，2002年。

道金斯，理查德，《自私的基因》，牛津大学出版社，2016年。

1996，Growing Artificial Societies，布鲁金斯学会出版社和麻省理工出版社。

费希尔，莱恩，《完美的蜂群》，基础书籍，2009年。

《自然的计算之美》，麻省理工学院出版社 2000 年版。

不完整性》，威廉·沃德尔·诺顿公司，2005年。

豹子如何改变它的斑点》，普林斯顿大学出版社，2001年。

- Holland, John, *Hidden Order*, Basic Books, 1995.
- Johnson, Steven, *Emergence*, Scribner, 2001.
- Kelly, Kevin, *Out of Control*, Basic Books, 2002.
- Kosko, Bart, *Fuzzy Thinking*, Hyperion, 1993.
- Levy, Steven *Artificial Life*, Pantheon, 1992.
- Mandelbrot, Benoit, *Fractal Geometry of Nature*, Times Books, 1982.
- McGrayne, Sharon Bertsch, *The Theory That Would Not Die*, Yale University Press, 2011.
- Mitchell, Melanie, *Complexity: A Guided Tour*. Oxford University Press, 2009.
- Waldrop, M. Mitchell *Complexity: The Emerging Science at the Edge of Order and Chaos*, Simon & Schuster, 1992.
- Resnick, Mitchell, *Turtles, Termites, and Traffic Jams*, Bradford, 1997.
- Rucker, Rudy, *The Lifebox, the Seashell, and the Soul*, Thunder's Mouth Press, 2005.
- Sawyer, R. Keith, *Social Emergence: Societies as Complex Systems*, Cambridge University Press, 2005.
- Schelling, Thomas, *Micromotives and Macrobehaviors*, W. W. Norton & Company, 2006.
- Strogatz, Steven, *Sync*, Hachette Books, 2003.
- Watts, Duncan, *Six Degrees*, W. W. Norton & Company, 2003.
- Wolfram, Stephen, *A New Kind Of Science*, Wolfram Media, 2002.

约翰，《隐藏的秩序》，基础书籍，1995。

史蒂文·约翰逊，《涌现》，斯克里布纳出版社，2001。

凯利，凯文，《失控》，Basic Books，2002。

模糊思考》，亥伯龙出版社，1993。

史蒂文·阿提社会生活》，万神殿，1992年。

曼德布洛特，伯努瓦，《自然的分形几何》，泰晤士出版社，1982。

不会消亡的理论》，耶鲁大学出版社，2011。

《复杂性: 导游》，牛津大学出版社，2009年。

复杂性: 处于秩序和混沌边缘的新兴科学》，西蒙与舒斯特，1992年。

雷斯尼克，米切尔，海龟，白蚁，和 Tra c Jams，布拉德福德，1997。

鲁克，鲁迪，《生命之盒》，《贝壳与灵魂》，雷霆出版社，2005年。

社会涌现: 作为复杂系统的社会，剑桥大学出版社，2005。

谢林，托马斯，《Micromotives 与宏观行为》，威廉·沃德尔·诺顿公司，2006。

斯托加茨，史蒂文，Sync，Hachette Books，2003。

威廉·沃德尔·诺顿公司，2003. Wolfram，Stephen，a New Kind Of Science，Wolfram Media，2002。



