R 语言统计编程

A First Course in Statistical Programming with R

W. John Braun, Duncan J. Murdoch 原著

赵时亮 译

R 语言统计编程

这是你所需要的唯一一本 R 语言编程入门书，R 是一个可免费下载的开放源代码语言，能让您编写满足您需求的程序。本书的合作方是 R 核心开发团队之一及专业 R 语言作者。本书配有规范的 R 语言代码。

与其他 R 语言入门书籍不同，本书着重于介绍如何有效编程，也包括能用于其它计算机语言的编程原理，以及用于开发更复杂项目的技巧。每个章节后的练习题能帮助读者更好更容易地学习本书，从而让读者更自信地学习。每章结束处提供了大量习题，它可帮助读者测试对学习内容的理解程度。

所有习题的答案、书中所用数据，以及勘误表都能从从本书的网站上查阅。

W. John Braun 是加拿大西安大略大学 (University of Western Ontario) 统计与精算系的副教授。他还曾与 John Maindonald 一起合著了《Data Analysis and Graphics Using R》。

Duncan J. Murdoch 也是西安大略大学 (University of Western Ontario) 统计与精算系的副教授。他曾于 1999-2000 年担任《Chance》杂志统计计算专栏的作者和编辑。

目 录

序言

本书的两位作者都任教于加拿大西安大略大学统计与精算系。最初，本书只是作为二年级学生上统计计算课（statistical computing）的笔记。我们对统计计算都很感兴趣，因为这有助于科研与教学，但在授课时我们发现，学生们通常都没有正确的编程基础。从本科生到博士生，他们都无法编写简单可靠的程序，他们不明白舍入错误如何能影响到数值计算的结果，他们也不知道如何开始一个较复杂的计算项目。

我们曾调查了其他系开设的相关课程，但那些课程更强调语言和概念，而这些内容对于统计专业的学生来说用处不大。我们的学生需要学一些简单的编程，使他们能运算一些仿真和随机模型；他们还需要了解数值分析，使他们可以做可靠的数值计算。由于我们无法找到能满足这些要求的课程，因此我们决定自己来做。

我们为本课程选择用 R 软件，因为 R 是一个开放源代码的软件，它在过去的几年中得到快速发展。开放源代码使得学生们能很容易获取并免费地安装到自己的电脑中。本书的作者之一 (Murdoch) 是 R 核心开发团队的成员之一，另一位作者 (Braun) 也是另一本书《Data Analysis Using R》的作者。这些因素决定我们选择 R 作为本书的软件。但我们都坚信，计算机编程具有一定的共性，我们也会在书中一再强调这些共性。不过本书不是关于 R 语言编程的手册，它只是关于 R 语言用于统计编程的教程。

学生在上这门课之前并不需要有编程的经验，也不需要掌握高级统计知识。但他们需要熟悉大学水平的微积分，需要了解一定的概率知识。关于概率的内容从第五章开始 (我们会给出一个简单的概率复习)。本书会包含一些较高级的内容，比如仿真，线性代数和优化等，如果本课程只教授一个学期，教师们可选择跳过这些较高级的部分。

我们感谢那些在本书写作过程中给予帮助的人们。感谢统计学 259b 班的学生，他们给了我们很大的鼓励并提供了很多有益的意见。感谢 Lutong Zhou 为本书绘制了大量图片。感谢剑桥大学出版社的 Diana Gillooly，牛津大学的 Brian Ripley 教授以及那些匿名评审者。如果没有 R 软件，也就不会有本书，如果没有全世界范围内 R 社区成员的贡献， R 软件也会逊色不少。

第 一 章

入门

欢迎来到统计编程的世界。本书包含许多关于 ``如何" 和 ``为什么" 等方面的专业意见。这一章首先会告诉您关于统计编程的基本概念，然后告诉您能在本书得到什么知识或技能，最后会指导您如何去下载和安装 R 软件，我们将利用这个软件和语言展示编程案例。

1.1 什么是统计编程?

计算机编程涉及到如何控制电脑，让电脑进行计算，以及显示计算结果等内容。什么是统计编程则比较难定义，有人说那是统计学家们所使用的计算程序，但统计学家使用过各种程序，哪一种才算？也有人说，统计编程就是人们在做统计时编写的程序，但统计过程包含大量的计算过程，到底哪一部分才是？

例如，统计学家们关注收集和分析数据，其中某些统计学家在工作中涉及到如何设置电脑与连接实验设备，不过这些都不能称之为统计编程。统计学家往往监督调查问卷的数据输入，并设置可接受的误差范围，以帮助检测数据输入的失误。这些工作可以称作统计程序，但他们太过专业，已超出了本书的范围。

统计编程涉及到通过计算来帮助进行统计分析。例如，数据首先必须被总结概括并显示，选用的模型必须符合数据特征，结果需要能恰当地显示出来。其实，很多应用程序都满足这样的要求，例如 Excel,SAS, SPSS, S-PLUS, R, Stata 等软都具备这样的功能。使用这些软件当然是在进行统计计算，而且有些软件通常还要求统计编程，不过本书关注的重点并不在此。在本书中，我们要提供理解那些应用程序工作的基础，描述它们如何进行计算，以及帮助读者如何通过编写代码来实现这些计算过程。

由于图形在统计分析中扮演很重要的角色，将一维、二维或更高维度的数据通过图形表达出来也是统计编程的一个方面。

统计编程中的一个重要部分是随机仿真 (stochastic simulation)。计算机本质上擅长于确定的、可再现的计算，但是现实世界中充满了随机过程。在随机仿真中，计算机扮演产生随机结果的角色。不过，如果我们对整个过程非常了解，那么结果还是明确的和可预见的。

统计编程和其他形式的数值计算密切相关，例如优化过程和数学函数逼近。不过，本书对差分方程关注较少，因为它们是物理学或应用数学中的重点。另外与纯计算机编程课程相比，我们更关注于计算结果而不是算法分析。

1.2 本书概要

本书是统计编程的入门书籍，我们会从基础编程开始，即怎样告诉电脑去执行一个操作。本书使用开放源代码软件 R 来实现这些操作，所以会教大家如何使用 R 软件，除此之外，还会强调那些在各种电脑编程平台上有共性的内容。

统计需要对数据进行透视，我们会告诉您如何构建统计图形，通过这些能学到关于人的视角及其如何影响人们选择数据透视的问题。

在介绍编程时，本书会告诉您如何控制一个可执行程序的运行。例如，您可能希望某个计算过程在输入值为正整数的时候一直持续下去，而当输入值达到 0 时停止计算。编写程序需要一定的逻辑思维，所以会接触一些布尔代数，这是正式的逻辑操作语句。最好的程序都在实施前需要经过深思熟虑，我们也将讨论如何把一个复杂的问题分解成简单的部分。当讨论编程时，将花费相当多的时间来讨论如何使它做得正确：如何确保这个程序在计算您想让它计算的问题。

统计编程一个显着的特征是它与随机性相关：数据及模型中如果包含随机部分就会包含随机误差。我们将讨论用特定的方式来模拟随机变量，并说明模拟随机变量在处理各种问题时是多么有用。

很多统计过程的基础是线性模型。但讨论线性回归和其他线性模型超出了本书的范围，不过我们确实会涉及到一些线性代数的基本知识，以及它们如何与计算过程相互关联。此外还会讨论最优化的一般问题：即搜寻使目标函数取得最大值或最小值的问题。

每一章之后，都会提供一定数量难度不一的练习题，读者可以从下面这个网站上找到习题答案[1]，http://www.stats.uwo.ca/faculty/braun/statprog/。

[1]本书已将习题答案附在书后

1.3 R 软件包

本书使用 R 软件包，这是一款能用于统计计算的开放源代码软件。开放源代码当然包含很多不同的意思，其中重要的一项就是能免费获取，使用者能不受限制地查看其源代码并进行再次开发。 R 软件包的基础是 S 语言，它是 1976 年贝尔实验室的 John Chambers 和其他一些合作者共同开发的。1993 年，奥克兰大学 (University of Auckland) 的 Robert Gentleman 和 Ross Ihaka 希望能用这种语言做一些实验，于是他们共同开发了一个可执行包，命名为 R 。在 1995 年他们公布了 R 语言的源代码，此后全球数百名爱好者对 R 语言的发展做出了贡献。

S 语言的商业应用包是 S-PLUS 程序，由于 R 和 S-PLUS 都是基于 S 语言开发的，所以不需要做什么修改，就能将 R 中的编写的程序移植到 S-PLUS 中。

1.4 为什么用命令行?

R 系统主要是命令驱动的，用户输入命令文本，并要求 R 来执行它。如今，大多数程序都使用交互式图形用户界面（菜单等）来执行程序操作，为什么本书选择这样一个老式的做法呢？

菜单型的操作界面在使用时非常方便，适用于有限的命令集，比如从几个命令到一两百个命令。然而，一个命令行界面是开放式。正如书中展示的那样，如果你想让计算机程序做一些前人没有做过的工作，您可以很容易地将整个任务分解成若干部分，然后用一个程序将它们组合起来。这在一些菜单型操作界面中或许也能做到，但显然在命令行界面中更容易实现。

此外，学习如何使用一个命令行界面会给您一些超过他人的技能，甚至可以让您更深入地理解菜单型操作界面是如何执行其操作的。作为统计人员，我们认为，您的目标应该是理解统计过程，学习如何在命令行界面中进行编程能满足您的这些基本要求。而学习使用基于菜单的程序则让您更依赖某些特定程序。

在 R 程序中有一个 Rcmdr 软件包，其中包含有相当丰富的菜单操作界面[2]。当您学习了这本书之后，如果您面对一个统计工作而不知道如何着手，那么您可能会发现在 Rcmdr 中的菜单为您提供了一些可供选择的想法。

1.5 字体约定

本书讲授如何用 R 进行计算，正如在后面的章节会看到的那样，用户需要键入一系列的文本命令作为输入，R 也会用一些文字或图像作为输出。为了在书中对输入输出的不同文字作出区分，用斜体字来表示用户的输入，用正常的直体字作为软件的输出[3]。例如：

> 这是用户输入

这是 R 所作出反馈（输出）

在除此之外的大部分案例中，我们会按实际的显示来表达 R 的输出[4]。

有时候，代码只是为了说明目的，并不意味着要执行[5]。在这些情况下，用正常的直体字来表示。例如：

f(some arguments)

1.6 安装 R 软件

用户可以从 R 项目的网站免费下载最新的软件包，其地址为http://cran.r-project.org/。大部分的用户需要下载并安装一个二进制的版本，这个版本已经被编译（被打包）成机器语言，可被用于特定操纵系统的特定类型的电脑执行。 R 具有很大的便携性，它可运行于 Microsoft Windows, Linux, Solaris, Mac OSX 等各种操作系统。不过要按不同的系统选择安装不同的二进制版本。本书中所编写的程序应该能适应不同的操作系统，不过我们假定读者使用的是 Microsoft Windows 。

在 Microsoft Windows 中安装 R 非常简单，一个可用于 Windows 98 或以上操作系统的二进制版本可从这个网页获取：http://cran.r-project.org/bin/windows/base。

下载一个名字为 "R-2.5.1-win32.exe" 的安装文件[6]，打开这个文件后，它就会几乎自动地进行安装。虽然用户可以进行自定义安装，不过在大多数情况下，特别是对于初学者来说哦，选择默认模式是最合适的。

---

[2]这个软件包是能在 R 中使用的若干功能程序的集合。

[3]译者注：翻译稿对此作了一定修改，输入的源代码用文本框显示，输出结果用带底纹的文字显示。

[4]因为使用了 Sweave 软件包，所以 R 本身会计算其输出。文中的输出都是使用 R 2.5.0 版本计算所得。

[5]有些代码不是正确的 R 代码，另外一些则是用来说明 R 的语法。

[6]译者： R 的版本更新很快，读者需要从网站上下载最新版本。

默认安装之后，会在桌面上创建一个 R 图标。一旦完成安装，您就可以开始学习统计编程了。下面就开始学习。

第 二 章

R 语言介绍

安装了 R 软件包之后，您就已经准备好学习统计编程的技艺了。第一步是学习所使用编程语言的语法，也就是了解这个语言的规则。本章介绍 R 语言的基本语法。

2.1 打开与退出 R

如果您是在 Microsoft Windows 系统中安装 R 软件，那么它会在开始菜单栏和桌面上创建快捷方式。双击 R 图标就能打开这个程序[1]。程序打开后会出现一个控制台界面，用户可以在这个界面中输入命令。

在控制台界面中会出现一个大于号 >，这是输入命令的提示符。看到这个提示符后，用户就可以键入命令了。例如可以将 R 作为一个计算器，在提示符后输入这样的数学表达式：

```
> 5+49
```

在按下 Enter 键之后，计算的结果 54 就出现了。在这个结果之前还有一个在方括号里面的前缀 1 。

```
[1] 54
```

这个 [1] 表示这是这个命令的第一个结果（仅限于这个例子）。其他命令可能会返回多个值，每一行输出结果前都会有一个标签，帮助用户来识别输出。例如用下面这样的方法来显示从 1 到 20 的一系列整数：

```
> options(width=40)
> 1:20
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
[13] 13 14 15 16 17 18 19 20
```

第一行第一列是这个命令的第一个返回值，所以在前面有标签 [1] ，而第二行的第一列是这个命令的第 13 个返回值, 所以前面的标签是 [13] 。所有能用袖珍计算器进行的计算都可以在 R 中实现。下面是一些例子：

```
> # "*" 星号是乘法符号
> # 井号之后是注释语句，会被忽略
> 3*5
```

```
[1] 15
```

```
> 3-8
```

[1] 其他操作系统可能需要点击不同的图标，或者会需要用户在命令行界面中键入 ``R'' 来打开软件。

```
[1] -5
```

```
> 12 / 4
```

```
[1] 3
```

要退出 R 进程，可以键入这个命令：

```
> q()
```

按确认键之后，会弹出一个对话框询问用户是否需要保存当前的工作环境，您可以选择保存或者不保存，也可以直接关闭这个窗口。工作空间中记录了用户所做的所有操作，甚至还包含一些已经保存的输出结果。如果用户选择取消退出 R 进程，那么就可以继续进行操作。我们一般很少会保存当前工作区，但偶尔会发现这样做也能带来一些便利。

请注意，如果在键入退出命令时忘了后面的括号会怎么样：

```
> q
```

```
function (save = "default", status = 0, runLast = TRUE)
.Internal(quit(save, status, runLast))
<environment: namespace:base>
```

之所以出现这种情况，是因为 q 是一个退出 R 的函数。如果键入 q 就相当于告诉 R 显示函数 q 的内容。而如果键入的是 q()，那么就是告诉 R 去执行函数 q()，即退出 R 进程。在 R 语言中，所有的操作都是通过调用函数来执行，不过有时候这个调用过程被隐藏起来了（比如通过点击菜单来执行操作），有时候这些函数太过基础也就不显示了，比如调用乘法函数来计算 $3 * 5$ 的时候。

2.1.1 记录工作

除了保存工作空间之外，人们更愿意保留一份输入命令的记录，这样在以后还能再现这个工作空间。在 Windows 操作系统下，最简单的方法是在 R 的脚本编辑器中输入命令，可以从文件菜单中打开这个编辑器。要执行某些命令时，可以选中这些命令，然后同时按下 Ctr-R 两个键（这相当于运行按钮）。在程序结束时，可以将所有的脚本保存为一个记录文件。在其他操作系统中，一个文本编辑器或使用类似剪切和粘贴等操作能达到相同的目的。

2.2 R 的基本操作

2.2.1 用 R 进行计算

从最基本的功能上看，R 可被看做是一个功能很全的计算器。从前一节中已经看到 R 可被用作数值计算，包括一些基本的操作，如 $+$ (加), $-$ (减), $*$ (乘), 和 $/$ (除)，还能用 ^ 来计算乘方。例如：

```
> 3^4
```

```
[1] 81
```

R 中也可以进行模算术。例如，可以计算 31 除以 7 之后的余数是多少，即 $31(mod 7)$。

```
> 31%%7
```

```
[1] 3
```

其中的整数部分是：

```
> 31 %/% 7
```

```
[1] 4
```

可以验证一下，整数部分乘以 7 之后，再加上余数，恰好就等于 31。

```
> 7*4+3
```

```
[1] 31
```

2.2.2 存储变量

R 有一个称为全局环境的工作空间，它可用来存储计算结果及其他各种对象。例如，我们希望存储 1.0025^30 的计算结果，以便以后能调用（这个表达式的含义是，假如投资的年利率为 0.25%，投资周期为 30 年，那么其复利为多少）。把这个计算结果赋值给目标变量 *interest*.30。在 R 中可以这样输入：

```
> interest.30 <- 1.0025^30
>
```

一个向左的箭头来告诉 R 进行赋值操作，这个向左的箭头可以用小于号 (<) 和减号 (−) 组成。当然 R 也支持使用等于号 (=) 来赋值，不过建议大家用箭头，因为用箭头可以很清晰地表示赋值操作，而用等于号的话看起来像是在表达一种关系（比如 *interest*.30 等于 1.0025^30），或者看起来像是做了一个永久的定义。请看，当按下确认键之后，屏幕上除了命令行提示符之外什么也没有显示，但其实 R 已经完成了用户请求的操作，并等着进一步的指令。如果需要显示这个变量的值，可以在提示符之后键入这个变量名：

```
> interest.30
```

```
[1] 1.077783
```

可以再做一个类似的计算，用 R 调用 *interest*.30 的值进行计算，并显示计算结果。而且以后随时都可以调用 *interest*.30 的值进行计算。例如计算投资 3000 美元，年利率为 0.25%，投资 30 年后的账户余额：

```
> initial.balance <- 3000
> final.balance <- initial.balance * interest.30
> final.balance
```

```
[1] 3233.35
```

例 2.1

假如某人从银行获得一笔贷款，贷款总额为 P, 利率为 i。在贷款一个月之后，此人打算用 n 个月来分期偿还，每月的还款额是 R。请问如何计算 R 的值？可以通过将 P 的现值与 n 个月，每月支付 R 的未来现金流贴现后相等来计算：

$P = R(1+i)^{-1} + R(1+i)^{-2} + \cdots + R(1+i)^{-n}.$

或者

$$P = R\sum_{j=1}^{n}(1+i)^{-j}.$$

这个几何级数求和并简化后，可以得到

$$P = R\left(\frac{1-(1+i)^{-n}}{i}\right).$$

这是一个年金现值公式，在给定了 P， n 和 i 之后，就可以求出 R 了。

$$R = P\frac{i}{1-(1+i)^{-n}}$$

在计算之前，首先定义几个变量：定义 P 为贷款额， *intRate* 为利率， n 为 分期付款月数，把每月应支付的额度赋值给变量 *payment*。

当然，还需要一些初始值来进行计算，假定贷款额是 1500 美元，利率为 1%，分 10 个月进行分期付款。用下面的代码来计算。

```
> intRate <- 0.01
> n <- 10
> principal <- 1500
> payment <- principal * intRate / (1 - (1 + intRate)^(-n))
> payment
```

```
[1] 158.3731
```

所以对于这笔贷款来说，每月的还款额应该是 158.37 美元。

2.2.3 函数

R 中的大部分工作都是通过调用函数来完成。例如前面已经说过退出 R 的命令是 q()，它告诉 R 去调用函数 q。括号中可以包含一些参数，在这个例子中参数是空：即只是告诉 R 退出，不需要告诉它怎么退出。我们可以看到 q 函数的定义是：

```
> q
```

```
function (save = "default", status = 0, runLast = TRUE)
.Internal(quit(save, status, runLast))
<environment: namespace:base>
```

这表明函数 q 有三个参数：分别为 ``save''，``status'' 和 ``runLast''。其中每个参数都有一个默认值：分别为 ``default''，``0''，和 ``true''。键入命令 q() 就是告诉 R 调用 q 函数，并以默认的参数来执行。

如果希望修改默认参数，那么在调用它时加以注明。例如：

```
> q("no")
```

```
> q(save = "no")
```

第一个命令告诉 R 在调用函数 q 时选择参数为 no，即退出 R 时不保存工作空间。如果设定两个参数但有没有命名的话，那么它们就会应用保存（save）和状态（status）两个参数。如果希望把前面的参数设为默认值，并希望以后还能对其进行修改，那么可以将其命名保存，并在以后调用这个函数时选择这个参数。

```
> q(runLast = FALSE)
```

或者可以用逗号表示缺失参数，例如：

```
> q( , , FALSE)
```

当调用的函数有好几个参数或者有一些不常用的参数时，一个好的习惯是给这些参数命名，这样在调用函数时就降低了出错的风险，也使代码更容易阅读。

2.2.4 精确还是近似？

计算中一个重要的区别在于结果是精确还是近似，本书中所做的大部分计算都采用近似模式。虽然可以用计算机来精确表达任何有理数，但更常见的是使用近似方式来表达：即通常所说的*浮点表达式*，它们是用二进制表达的科学记数法。例如，用科学计数法表达一个四位有效数的数字时，可以这样写 6.926×10^{-4}。这种计数方法可以表达在 0.00069255 和 0.00069265 之间的任何一个数字。计算机中标准的浮点表达式和这个类似，只不过基底用的是 2 而不是 10 。分数也可以用二进制来表达。上面这个数字用二进制可以写为 $1.011_2 \times 2^{-11}$。下标 2 表示这是以 2 为底的数字，即它表示 $1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$，或者用十进制表示的 1.375。

然而，6.926×10^{-4} 和 $1.011_2 \times 2^{-11}$ 这两个数字是不一样的。四位的二进制表达式其精度要低于四位的十进制表达式。对于四位二进制表达式来说，它表达从 0.000641 到 0.000702 时都是一样的。事实上，6.926×10^{-4} 并不能用有限个二进制数来精确表达。这个问题类似于用十进制来表示 1/3 这个分数，0.3333 可能是一个很好的近似，但它并不是一个精确的数字。在 R 中用 53 位的二进制表达式，相当于十进制中 15 到 16 位的数字。

考虑两个分数 5/4 和 4/5，如果用十进制来表示上面两个数字的话，分别为 1.25 和 0.8 。5/4 用二进制表示的话是 $1 + 1/4 = 1.01_2$ 。那 4/5 用二进制该如何表示呢？因为它处于 0 和 1 之间，所以其形式应该是 $0.b_1b_2b_3\cdots$，其中每一个 b_i 代表一个 ``比特''，即一个 0 或者一个 1 。乘以 2 之后所有的数字都往左移一位，即 $2 \times 4/5 = 1.6 = b_1.b_2b_3\cdots$ 。这样 $b1 = 1$，而 $0.6 = 0.b_2b_3$ 。

现在将上式再乘以 2，即 $2 \times 0.6 = 1.2 = b_2.b_3\cdots$，这样可知 $b_2 = 1$ 。将上面这个过程重复两次可得 $b_3 = b_4 = 0$，读者可以自己尝试一下。

到了这一步，又得到数字 0.8，所以前面 4 个比特的数字会重复出现，因此知，用二进制表示 4/5 的话，应该是 $0.110011001100\cdots$ 。由于 R 只精确地表达 53 位数字，所以它不可能精确地存储 0.8 这个数字，其中必然包含一些舍入误差。

可以用实验来测试这个舍入误差，从理论上分析可知 $(5/4) \times (4/5) = 1$，所以 $(5/4) \times (n \times 4/5)$ 应该精确地等于 n，但如果在 R 中来计算的话，其结果是这样的：

```
> n <- 1:10
> 1.25 * (n * 0.8) - n
```

```
[1] 0.000000e+00 0.000000e+00 4.440892e-16 0.000000e+00 0.000000e+00
[6] 8.881784e-16 8.881784e-16 0.000000e+00 0.000000e+00 0.000000e+00
```

它们都等于某个值，而且当 $n = 3$，6 和 7 时，结果还显著地不相同。不过这些误差项都很小，但绝不等于零。

舍入误差在很多计算中会累积起来，所以经过一些列的计算之后，其误差可能会很大。有些操作也特别容易产生舍入误差：例如，两个几乎相等的数字相减，或等价地来说两个符号相反绝对值近似的数字相加。因为对于这两个近似的数字来说，在二进制表达中，其第一位的数字是一样的，所以他们相减时就消去了，最后的结果是由存储位置靠后的那些数字决定的。

例 2.2

考虑某个抽样样本的方差， $x_1, \cdots, x_n$ ：

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

其中 $\bar{x}$ 是样本均值，它等于 $(1/n)\sum x_i$ 。在 R 中，s^2 可以用 $var()$ 函数获取，$\bar{x}$ 可以用 $mean()$ 函数获取。例如：

```
> x <- 1:11
> mean(x)
```

```
[1] 6
```

```
> var(x)
```

```
[1] 11
```

```
> sum( (x - mean(x))^2 ) / 10
```

```
[1] 11
```

因为这个公式首先需要计算 $\bar{x}$，然后再计算离差的平方和，所以在计算过程中要求将所有的 x_i 调到内存中。在不久以前，内存还是很昂贵的，所以为了节约内存，就将公式改写成这个样子：

$$s^2 = \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2\right)$$

这个公式被称作"one-pass formula"，因为每个 x_i 只执行一次，然后将 x_i 和 x_i^2 累加起来。对于我们所举的例子，他们从数学上都给出了正确的答案。

```
> ( sum(x^2) - 11 * mean(x)^2) / 10
```

```
[1] 11
```

然而，如果给每个 x_i 都加上一个很大的数 A 之后，$\sum_{i=1}^{n} x_i^2$ 的值大约增加了 nA^2，同样 $n\bar{x}^2$ 也相应地增加。这个过程不会改变样本的方差，但当求他们的差时，它就带来``灾难性的精度损失"。

```
> A <- 1.e10
> x <- 1:11 + A
> var(x)
```

```
[1] 11
```

```
> ( sum(x^2) - 11 * mean(x)^2) / 10
```

```
[1] 0
```

既然 R 给出了正确的答案，那么它显然就没有使用 the one-pass formula，所以您也不应该使用这个公式。

2.2.5 R 区分大小写

请键入下面这个命令，看看有何结果。

```
> x <- 1:10
> MEAN(x)
```

其输出为：

```
Error: could not find function "MEAN"
```

如果改成这样的语句就可以正确显示了。

```
> x <- 1:10
> mean(x)
```

```
[1] 5.5
```

现在您可以试试这样命令：

```
> x <- 1:10
> MEAN <- mean
> MEAN(x)
```

```
[1] 5.5
```

mean() 是 R 中的内部函数，而 R 会认为 MEAN 是一个不同的函数，因为 R 的命令中是区分大小写的，比如 m 就是和 M 是两个完全不同的字母。

2.2.6 列出工作空间中的对象

前面的各种计算会创建若干对象，这些对象被储存在 R 工作空间中。可以用函数 objects() 来列出当前工作空间中所有的对象，并显示在屏幕上。

```
> objects()
```

```
[1] "A"            "final.balance"  "initial.balance"
[4] "interest.30"  "intRate"        "MEAN"
[7] "n"            "payment"        "principal"
[10] "saveopt"     "x"
```

一个与 objects() 函数功能相同的函数是 ls() 。

前面说过，如果退出 R 时选择不保存当前工作空间，所有的对象都会被丢弃。如果保存工作空间，那么当下一次启动 R 时，就会加载它。[2]

2.2.7 向量

一个数值向量是一系列的数字。用 c() 函数可以将各种对象存储到向量中，比如可以这样输入：

```
> c(0, 7, 8)
```

```
[1] 0 7 8
```

当然，还可以将这个向量存储到一个对象中：

```
> x <- c(0, 7, 8) # x 是包含三个元素的向量
```

为了显示对象 x 中的内容，只需要键入 x 就行。

```
> x
```

[2] 一般说来如果从上一次结束 R 的同一个文件夹再次启动 R，被保存的工作空间都能自动加载。不过用户可以通过菜单操作或用命令 setwd() 来改变当前的工作目录。

```
[1] 0 7 8
```

使用冒号 (:) 可以创建一个数值序列（升序或者降序），例如：

```
> numbers5to20 <- 5:20
> numbers5to20
```

```
 [1]  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

使用 c 函数，可以将几个向量合并到一起。例如下面的命令就是将一个序列和 x 两个向量合并为一个向量。

```
> c(numbers5to20, x)
```

```
 [1]  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20  0  7  8
```

这是使用 c() 函数的另一个例子：

```
> some.numbers <- c(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
+ 43, 47, 59, 67, 71, 73, 79, 83, 89, 97, 103, 107, 109, 113, 119)
```

请注意，R 用一个加号 (+) 提示用户第二行的输入，这是因为第一行的输入并没有完成。我们可以在 *some.numbers* 变量之后添加 *numbers5to20* 变量，最后再添加从 4 到 1 的一个递减顺序：

```
> a.mess <- c(some.numbers, numbers5to20, 4:1)
> a.mess
```

```
 [1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43 47 59 67 71 73
[20] 79 83 89 97 103 107 109 113 119  5  6  7  8  9 10 11 12 13 14
[39] 15 16 17 18 19 20  4  3  2  1
```

前面说过，方括号内的数字表示的是输出索引，通过它可以很容易地指出 *a.mess* 中的第 22 个元素是 89。

2.2.8 从向量中提取元素

可以用一个更便捷的方式来显示 *a.mess* 中的第 22 个元素，只需要输入这样的命令即可：

```
> a.mess[22]
```

```
[1] 89
```

要显示 x 变量中的第二个元素，只需要输入这样的命令：

```
> x[2]
```

```
[1] 7
```

也可以同时提取多个元素，例如：

```
> some.numbers[c(3, 6, 7)]
```

```
[1]  5 13 17
```

还可以同时提取从第三个直到第七个元素，例如：

```
> numbers5to20[3:7]
```

```
[1]  7  8  9 10 11
```

如果用一个负数，那就表示除此之外的其他元素。例如，我们希望列出 x 中除第二个元素外的所有元素：

```
> x[-2]
```

```
[1] 0 8
```

同样，如果希望将第 3 到第 11 个元素排除在外，就可以这样写命令：

```
> some.numbers[-(3:11)]
```

```
 [1] 2 3 37 41 43 47 59 67 71 73 79 83 89 97 103 107
[17] 109 113 119
```

用零作为索引的话，就什么都不返回。虽然通常不会这么做，不过有时候面对一个复杂表达式时，采用这个方法还是很有用的。

```
> numbers5to20[c(0, 3:7)]
```

```
[1]  7  8  9 10 11
```

不过读者需要注意，不能将正的和负的索引同时引用，例如下面的命令就会出错。

```
> x[c(-2, 3)]
```

```
Error: only 0's may be mixed with negative subscripts
```

原因在于同时使用正和负的索引使表达式逻辑上不清晰，是要列出除去第二个元素之后的第三个还是除去第二个元素之前的第三个呢？

2.2.9 向量运算

在 R 中也可以进行向量运算，比如可以将向量 x 中的所有元素都乘以 3：

```
>x*3
```

```
[1] 0 21 24
```

请注意，计算是在元素层面上进行的，如果加、减或除一个常数的话，结果也大致相同，比如：

```
>y <- x-5
>y
```

```
[1] -5  2  3
```

请再看对每个元素求三次方的运算：

```
> x^3
```

```
[1]   0 343 512
```

上面的例子显示了一个二进制算术运算符如何应用于向量和常数。一般说来，这些运算符也同样适用于一对向量之间进行元素对元素的计算。例如，可以计算 $y_i^{x_i}$，其中 $i = 1, 2, 3$，即 $y_1^{x_1}, y_2^{x_2}, y_3^{x_3}$。

```
> y^x
```

```
[1]   1  128 6561
```

当向量的长度不同时，那么短的那个向量就会被循环选取。例如要计算从 1 到 10 这十个数字分别对 2 和 3 求模后的余数，那么只需要输入一次向量 [2,3] 就够了。

```
> c(1, 1, 2, 2, 3, 3, 4, 4, 5, 5,
+ 6,6,7,7,8,8,9,9,10,10)%%2:3
```

```
 [1] 1 1 0 2 1 0 0 1 1 2 0 0 1 1 0 2 1 0 0 1
```

如果长向量的长度不等于短向量的整数倍，那么 R 就会给出警告信息。例如把上例中的短向量改成 [2,3,4]，那么就会有这样的出错信息。

```
> c(1, 1, 2, 2, 3, 3, 4, 4, 5, 5,
+ 6,6,7,7,8,8,9,9,10,10)%%2:4
```

```
 [1] 1 1 2 0 0 3 0 1 1 1 0 2 1 1 0 0 0 1 0 1
Warning message:
In c(1, 1, 2, 2, 3, 3, 4, 4, 5, 5, +6, 6, 7, 7, 8, 8, 9, 9, 10,  :
  longer object length is not a multiple of shorter object length
```

2.2.10 创建简单格式的向量

我们已经知道用冒号可以产生一个序列的整数。除此之外，还可以用 seq() 函数和 rep() 函数来产生某种格式化的向量。例如，要生成小于等于 21 的所有奇数序列，可以用这一命令：

```
seq(1, 21, by=2)
```

请注意，在这里用了 by=2 这样的选项。seq() 函数有好几个选项，其中一个就是 by 。如果没有加上 by 选项，那么默认值就是 1 。要产生一个元素重复的向量，可以使用 rep() 函数。请看下面这些例子：

```
> rep(3, 12) # 把 3 重复 12 次
```

```
 [1] 3 3 3 3 3 3 3 3 3 3 3 3
```

```
> rep(seq(2, 20, by=2), 2) # 把 2 4 ... 20 这样的格式重复 2 次
```

```
[1] 2 4 6 8 10 12 14 16 18 20 2 4 6 8 10 12 14 16 18 20
```

```
> rep(c(1, 4), c(3, 2)) # 把 1 重复 3 次，把 4 重复 2 次
```

```
[1] 1 1 1 4 4
```

```
> rep(c(1, 4), each=3) # 把每个值都重复 3 次
```

```
[1] 1 1 1 4 4 4
```

```
> rep(seq(2, 20, 2), rep(2, 10)) # 把每个值重复 2 次
```

```
[1] 2 2 4 4 6 6 8 8 10 10 12 12 14 14 16 16 18 18 20 20
```

2.2.11 缺失值及其他特殊数值

在 R 中，缺失值用 NA 表示。缺失值通常来自于现实中一些有问题的数据，不过有时候在计算过程中也会产生一些缺失值。

```
> some.evens <- NULL # 创建一个空向量
> some.evens[seq(2, 20, 2)] <- seq(2, 20, 2)
> some.evens
```

```
[1] NA 2 NA 4 NA 6 NA 8 NA 10 NA 12 NA 14 NA 16 NA 18 NA 20
```

在上面的命令中，给偶数位置赋 $2, 4, \cdots, 20$ 这样的值，但对奇数位置不赋值。这样 R 就自动给这些数值未知的地方赋 NA 这个值。回过头来，再看 x 这个向量，它的值是 $(0, 7, 8)$，那么下面的计算会出现什么情况呢？

```
> x / x
```

```
[1] NaN 1 1
```

这里 NaN 表示 ``这不是一个数字"，因为在上述命令试图去计算 0/0 。有时候 NaN 也用来表示计算过程不合理。在某些情况下，会显示一些特殊的值，或者得到错误警告。

```
> 1 / x
```

```
[1] Inf 0.1428571 0.1250000
```

这里 R 试图计算 1/0 。所以结果用 Inf 表示。

读者必须注意，向量的索引值必须是整数，如果试图使用分数来做向量的索引，那么它会被截断为 0 ，比如下面这样输入的 0.4 就被截断为 0 。`

```
> x[0.4]
```

```
numeric(0)
```

这里的输出 $numeric(0)$ 表示一个长度为 0 的数值向量。

2.2.12 字符串向量

标量和向量除了可以用数字表示外，还可以用字符串表示。不过一个向量中所有元素都必须是同一类型。例如：

```
> colors <- c("red", "yellow", "blue")
> more.colors <- c(colors, "green", "magenta", "cyan")
> z <- c("red", "green", 1) # 给向量赋不同类型的值
```

想要查看 $more.colors$ 和 z 的内容，只需要键入如下命令：

```
> more.colors
```

```
[1] ”red” ”yellow” ”blue” ”green” ”magenta” ”cyan”
```

```
> z # 其中数字 1 转换成了字符串类型
```

```
[1] ”red” ”green” ”1”
```

对字符串向量希望能执行两个基本操作。第一个基本操作是提取一个子字符串，使用 substr() 函数可以实现这个功能。其格式为 substr(x, start, stop) ，其中 x 是字符串向量， $start$ 和 $stop$ 告诉 R 哪些是需要保留的。例如输出 $color$ 向量中前面两个字母：

```
> substr(colors, 1, 2)
```

```
[1] "re" "ye" "bl"
```

另外一个函数 substring() 也具有类似的功能，不过使用时稍有不同，具体可以用 ?substring 查看帮助。另一个基本的操作是通过连接字符串来产生新的字符串向量。这个功能可以用 paste() 来实现。例如:

```
> paste(colors, "flowers")
```

```
[1] "red flowers" "yellow flowers" "blue flowers"
```

在 paste() 函数中有两个选项，其中 *sep* 参数控制两个字符串合并时中间是否相互分割。如果不希望合并时保留中间的空格，就可以用下面的选项：

```
> paste("several", colors, "s", sep="")
```

```
[1] "several reds" "several yellows" "several blues"
```

另一个选项 *collapse* 则将合并后向量的所有部分塌缩成单一字符串：

```
> paste("I␣like", colors, collapse = ",␣")
```

```
[1] "I like red, I like yellow, I like blue"
```

2.2.13 因子

因子提供了另一种存储字符数据的方法。例如，可以用下面的命令来创建一个包含四个元素两个层次的 control 和 treatment 字符串：

```
> grp <- c("control", "treatment", "control", "treatment")
> grp
```

```
[1] "control" "treatment" "control" "treatment"
```

```
> grp <- factor(grp)
> grp
```

```
[1] control treatment control treatment
Levels: control treatment
```

对于那些有重复元素的向量，“因子”是一个很有效的存储字符变量的方式。这是因为在内部是用整数来保存因子的层次。如果想看看这个因子的代码，可以这样输入：

```
> as.integer(grp)
```

```
[1] 1 2 1 2
```

层次的标签只被存储一次，而不是重复存储。代码作为指数进入向量的层次。

```
> levels(grp)
```

```
[1] "control" "treatment"
```

```
> levels(grp)[as.integer(grp)]
```

```
[1] "control" "treatment" "control" "treatment"
```

2.2.14 从向量中提取元素

和数值向量一样，R 也用方括号 [] 来表达字符串元素的索引。例如因子 *grp* 有四个元素，所以可以用这样的命令来输出第三个元素：

```
> grp[3]
```

```
[1] control
Levels: control treatment
```

也可以用下面的命令来输出 *more.colors* 向量中从第二个到第五个元素。

```
> more.colors[2:5]
```

```
[1] "yellow" "blue" "green" "magenta"
```

可以用 is.na() 函数来侦测向量中是否包含缺失元素，例如：

```
> is.na(some.evens)
```

```
 [1] TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
[12] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE
```

输出结果是一个``逻辑向量"。我们在 2.4 节中会做更多的介绍。
符号 ! 用来表示逻辑``非"，这样就可以列出 *some.evens* 向量中的非缺失值：

```
> !is.na(some.evens)
```

```
 [1] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE
[12] TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
```

也可以列出其中的偶数值：

```
> some.evens[!is.na(some.evens)]
```

```
[1] 2 4 6 8 10 12 14 16 18 20
```

2.2.15 矩阵与数组

要将输入的数字排列为矩阵，可以使用 matrix() 函数：

```
> m <- matrix(1:6, nrow=2, ncol=3)
> m
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

我们可以使用两个索引参数来读取某一个元素。例如，读取矩阵中第一行第二列的元素，可以用这样的命令：

```
>m[1, 2]
```

```
[1] 3
```

可能有些混淆的是，R 同时也允许像向量一样来对矩阵元素进行索引，此时只需要用一个索引值就行了：

```
> m[4]
```

```
[1] 4
```

这时候，元素是按照他们存储时候的顺序被检索的，其顺序是先按第一列从上到下，然后是第二列从上到下检索，直到最后一列。这被称为以列为主的存储顺序。一些其他的计算机软件可能采用按行为主的存储顺序，即存储顺序是先从第一行从左到右，然后是第二行，直到最后一行。

如果要列出整个一行或一列的元素，可以用这样的命令：

```
> m[1,]
```

```
[1] 1 3 5
```

```
> m[, 1]
```

```
[1] 1 2
```

一个更常见的数据存储方式是使用数组。数组有多维指数，可以使用数组函数来创建：

```
> a <- array(1:24, c(3, 4, 2))
> a
```

```
, , 1
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
, , 2
     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

请注意，我们使用了一个向量 $c(3,4,2)$ 来指定数组的维度。当插入数字之后，第一个索引变量是最快的，当它全部运行一遍之后，第二个索引变量改变了，以此类推。

2.2.16 数据框

在 R 中存储的大部分数据都采用数据框的方式。它们类似于矩阵，但它们每一列都有自己的名字，列与列之间也可以是不同的类型。使用 data.frame() 函数可以创建这样的结构化数据。

从向量创建数据框：

```
> colors <- c("red", "yellow", "blue")
> numbers <- c(1, 2, 3)
> colors.and.numbers <- data.frame(colors, numbers,
+ more.numbers=c(4, 5, 6))
```

可以查看一下数据框的内容：

```
> colors.and.numbers
```

```
  colors numbers more.numbers
1    red       1            1
2 yellow       2            2
3   blue       3            3
```

习题

1. 计算 170166719 除以 31079 之后的余数。
2. 已知有一笔贷款额为 $ 200000，每月的利率为 0.003，还款周期为 300 个月。请计算每月的还款额度。
3. 计算下列式子，$\sum_{j=1}^{n} r^j$，其中 R 的值为 1.08，n 的值分别选 10，20，30 和 40。然后再用这个公式 $\frac{1-r^{n+1}}{1-r}$ 计算一次，比较两种方法的计算结果。最后将 R 的值换成 1.06 再重复一次上面的计算。
4. 用简化的公式计算 $\sum_{j=1}^{n} r^j$，设 $r = 1.08$，计算当 n 从 1 变化到 100 时的值，并将这个值储存为一个向量。
5. 分别用两种方法计算，当 $n = 100, 200, 400, 800$ 时 $\sum_{j=1}^{n} j$ 和 $\frac{n(n+1)}{2}$ 的值有何区别。
6. 用简化的公式计算 $\sum_{j=1}^{n} j$，当 n 从 1 变化到 100 时的结果，并将它储存为一个向量。
7. 分别用两种方法计算，当 $n = 200, 400, 600, 800$ 时 $\sum_{j=1}^{n} j^2$ 和 $\frac{n(n+1)(2n+1)}{6}$ 的值有何区别。
8. 用简化的公式计算 $\sum_{j=1}^{n} j^2$，计算当 n 从 1 变化到 100 时的值，并将这个值储存为一个向量。
9. 计算下列公式的结果：$\sum_{i=1}^{n} 1/i$，其中 n 分别取 500，1000，2000，4000，8000。然后再用 $\log n + 0.6$ 的公式再计算一遍。比较两种方式答案的异同。
10. 试解释下面输出的结果（参阅 2.2.4 节）：

    > x <- c(0,7,8)

    > x[0.9999999999999999]

    numeric(0)

> x[0.99999999999999999]

[1] 0

11. 使用 rep() 或者 seq() 函数，创建下面的向量。

0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4

和

1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5

12. 使用 rep() 或者 seq() 函数，创建下面的向量。

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

13. 使用 more.colors 向量，以及 rep() 或者 seq() 函数，创建下面的向量。

”red” ”yellow” ”blue” ”yellow” ”blue” ”green”

”blue” ”green” ”magenta” ”green” ”magenta” ”cyan”

2.2.17 日期与时间

使用计算机来处理日期型和时间型数据是最困难的任务之一。标准的日历很复杂，每个月的长度不同，每四年还有一个闰年（每世纪有一次例外）。当要查看某段历史时期，比如从儒略历转换到现代的格里高利历时，就会遇到各种问题，因为从 1582 年到 1923 年间，不同的国家采用的日历不同。这样解释日期的时候就会有困难。

时间有时候也会很混乱，因为有些地区会采用夏令时或冬令时，在有些年份还会因为地球自转因素出现闰秒等现象。

在 R 中有多个函数可以处理时间和日期。例如用 strptime() 函数可以将字符串（如 $2007-12-25$, 或者 12/25/07）转换为内部时间数字表达式。用 format() 函数可以将内部时间表达式转换为常见的日期时间表达式。使用 ISOdate() 和 ISOdatetime() 函数将日期和时间分别用标准格式表示。以上这些函数都包含在 R 的基本安装程序包中，而在 chron 程序包中还包含更多的日期时间程序，读者可以参阅其他参考资料。

2.3 内建函数与在线帮助

函数 q() 是一个内建函数的例子。在 R 中有很多这样的内建函数，具有各种功能。在线的帮助文件能协助用户了解这些函数的功能。用户有多种方式可以获取帮助。假如用户知道函数的名字，就可以使用 help() 函数。假如用户想了解 q() 函数的相关资料，可以输入：

```
> ?q
```

或者

```
> help(q)
```

上面命令中的任何一个都会打开一个新窗口，并在其中列出关于这个函数的帮助信息。

在 R 中，另一个经常使用的函数是 mean()。

```
> help(mean)
```

该命令会打开一个新窗口，它的内容包括 (不同的操作系统，显示内容稍有差别)：

mean package:base R Documentation

Arithmetic Mean

Description:

Generic function for the (trimmed) arithmetic mean.

Usage:

mean(x, ...)

Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)

Arguments:

x: An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects, and for data frames all of whose columns have a method. Complex vectors are allowed for 'trim = 0', only.

trim: the fraction (0 to 0.5) of observations to be trimmed from each end of 'x' before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.

这些说明表示 mean() 函数将计算变量的算术平均值，还能根据用户的要求进行 ``微调"。
如果要计算变量 x 的均值，只需要键入：

```
> mean(x)
```

[1] 5

2.3.1 内建例子

与 help() 函数类似，用户还可以用 example() 函数来获取帮助。

```
> example(mean)
```

```
mean> x <- c(0:10, 50)

mean> xm <- mean(x)

mean> c(xm, mean(x, trim = 0.10))
[1] 8.75 5.50

mean> mean(USArrests, trim = 0.2)
  Murder   Assault UrbanPop     Rape
    7.42    167.60    66.20    20.16
```

这个例子显示了 mean() 函数如何工作，以及如何使用 trim 选项做微调。（如果 $trim = 0.1$，在计算均值时，最高和最低 10% 的数据被删除）

2.3.2 利用帮助文件

最常用也是最方便的方式是使用 help.star() 函数。这个函数会打开网络浏览器，比如 Internet Explorer 或者 Firefox 。浏览器中会显示一个菜单和若干选项，包括安装软件包的列表。基本安装包中包含了很多常用功能。

另一个有用的函数是 help.search()，假如用户想了解哪些函数能进行最优化计算（求极大值和极小值），那么可以这样输入：

```
help.search("optimization")
```

其显示结果为：

```
Help files with alias or concept or title matching ‘optimization’
using fuzzy matching:

maxLik::returnCode      Return code for optimisation and other
                        objects
maxLik::returnMessage
                        Information about the optimisation process
nlme::lmeScale          Scale for lme Optimization
```

```
rgenoud::genoud        GENetic Optimization Using Derivatives
stats::constrOptim     Linearly constrained optimisation
stats::nlm             Non-Linear Minimization
stats::nlminb          Optimization using PORT routines
stats::optim           General-purpose Optimization
stats::optimize        One Dimensional Optimization
tseries::portfolio.optim
                       Portfolio Optimization
Zelig::model.end       Cleaning up after optimization



Type '?PKG::FOO' to inspect entry 'PKG::FOO TITLE'.
```

根据前面的帮助内容，用户可以进一步查看 nlm() 函数的帮助文件。

```
help(nlm)
```

使用类似 Google 这样的网络搜索引擎，也能得到 R 的很多帮助。在搜索关键词中包含 ``R'' 的话，通常会显示相关的帮助内容。

还有一个有用的函数是 RSiteSearch()，这个函数在 R 的邮件列表中搜寻帮助。例如，想知道 R 如何处理缺失记录的话，可以这样搜索：

```
RSiteSearch("missing")
```

2.3.3 内建图形函数

统计分析中，最常用的是绘制直方图和散点图。请看下面的例子：

```
> x <- c(12, 15, 13, 20, 14, 16, 10, 10, 8, 15)
> hist(x)
```

```
> x <- seq(1, 10)
> y <-x^2-10*x
> plot(x, y)
```

请看，在图 (2.2) 中，x 的值作为水平轴。

另一个比较有用的绘图函数是 curve()，它用来绘制在指定区间内某数学函数的曲线，分别用 *from* 和 *to* 来指定区间。下面用一个简单例子来显示在 $[0\ 6\pi]$ 之间的正弦曲线。

```
> curve(expr = sin, from = 0, to=6*pi)
```

在这个函数中，*expr* 参数可以是一个函数（当输入一个数字向量时，能输出一个数字向量），也可以是一个关于变量 x 的表达式。比如可以用这样的表达式来作图：

```
curve(x^2 - 10 * x, from = 1, to = 10)
```

我们将在第三章中介绍更多的绘图函数及应用。

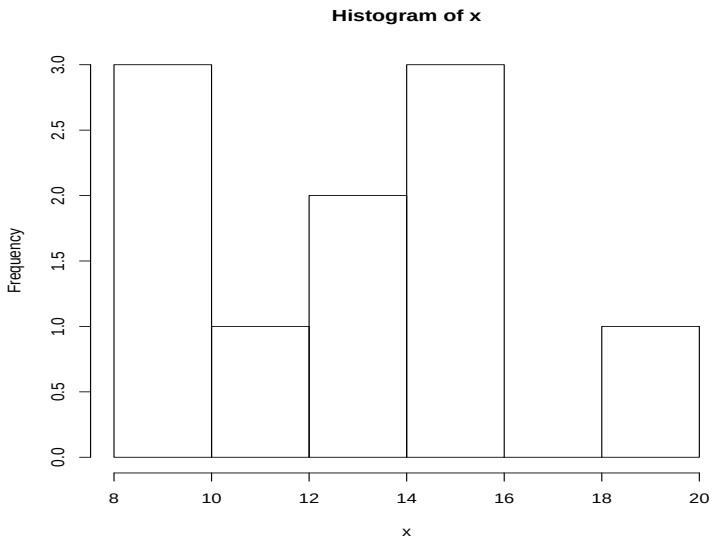


图 2.1: 一个简单的直方图

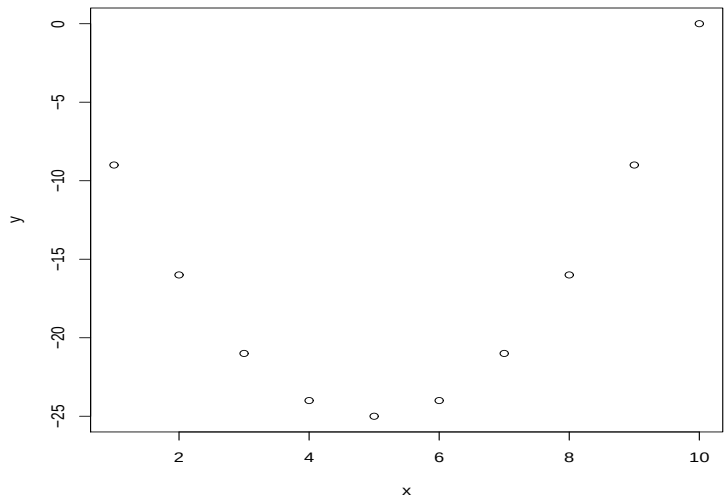


图 2.2: 一个简单的散点图

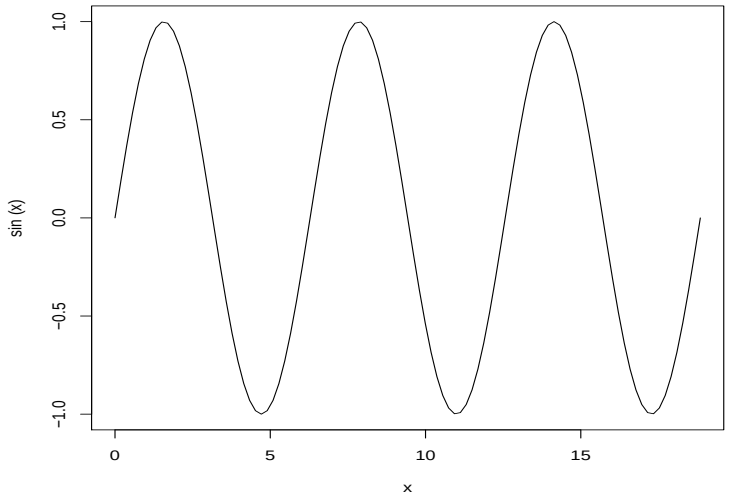


图 2.3: 正弦曲线

2.3.4 附加内建函数

样本中位数

样本中位数表示一个数据集中位于最中间的值。如果数据是 $x[1] \leq x[2] \leq \cdots \leq x[n]$，当 n 是奇数时，那么中位数就是 $x[(n+1)/2]$，当 n 是偶数时，那么中位数就是 $\{[x[n/2] + x[n/2+1]\}/2$。

例如 $10, 10, 18, 30, 32$ 的中位数是 18，而 $40, 10, 10, 18, 30, 32$ 的中位数是 18 和 30 的均值，即 24。在 R 中计算中位数可以用这样的函数：

```
> median(x)
```

其他描述性统计函数

用描述性统计函数可以揭示数据向量中的统计特征。例如可以尝试这些函数：

```
var(x) # 计算 x 的方差
summary(x) # 给出 x 的统计摘要
```

习题

1. 下面是一个暖房中接受太阳辐射观察值的样本数据。$11.1, 10.6, 6.3, 8.8, 10.7, 11.2, 8.9, 12.2$

   - 把数据赋值给一个名叫 *solar.radiation* 的变量。
   - 计算这个数据的均值，中位数和方差。
   - 给上述数据的每个观察值加 10，并赋值给变量 $sr10$。计算 $sr10$ 的均值，中位数和方差。哪些统计值改变了？改变了多少？
   - 每个观察值乘以 (-2)，并将结果赋值给变量 $srm2$。计算 $srm2$ 的均值，中位数和方差。哪些统计值改变了？改变了多少？
   - 绘制 *solar.radiation*, $sr10$ 和 $srm2$. 的直方图。
   - 通常有两种方法来计算方差，分别是 $1/n\sum_{i=1}^{n}(x_i - \bar{x})^2$ 和 $1/(n-1)\sum_{i=1}^{n}(x_i - \bar{x})^2$。其中一个用 n，另一个用 $(n-1)$。请问，在 R 中用的是哪一个公式？

2.4 逻辑向量和关系运算

前面已经学习使用了 c() 函数来给数字向量和字符串向量赋值。 R 同样也支持逻辑向量。它们包含两个单元 ``是"(TRUE) 与 ``否"(FALSE)。

2.4.1 布尔代数

要理解 R 怎么处理 $TRUE$ 和 $FALSE$ ，就必须懂得一些 ``布尔代数"。布尔代数的主要内容是使用数学方法来解决逻辑问题。

逻辑用来处理一些要么为真，要么为假的陈述。我们用一个字母或变量来代替每个陈述，比如， A 代表天很晴朗， B 代表天正下雨。根据您当地的天气，这两个陈述可能都对，比如正好有太阳雨。在某个普通的晴天 A 可能对， B 可能错，而在某个多雨的季节， B 可能对， A 可能错。也有可能两者都错，比如恰好是一个多云但干燥的天气。

布尔代数告诉我们如何去评估一个复合句的对与错。比如 `` $A$ 与 $B$ " 就是一个复合句，它表示即晴朗又下雨。这个陈述只有在太阳雨的情况下才对。`` A 或者 B " 表达的是天气或者晴朗，或者在下雨，或者既晴朗又下雨，即表示除了多云干燥的天气之外的所有情况。而 `` $A$ 异或 $B$ "，它表示要么是晴天，要么是下雨，但肯定不是两者同时成立。还有一个关系是表示否定关系，可以表示为 ``非 A "，它表示所有不是晴天的情况。

在布尔代数和集合论之间有一个很重要的关系。如果把 A 和 B 看做集合，那么可以将 `` $A$ 与 $B$ " 看做一个集合其中的元素既包含 $A$ 又包含 $B$ ，即集合 $A$ 与 $B$ 的交集 $A \cap B$ 。与此类似， $A$ 或 $B$ 也可以表示成一个集合，其元素要么在 $A$ 中，要么在 $B$ 中, 可以用并集 $A \cup B$ 表示。最后，``非 A " 就是 A 的补集。

因为只有两个可能值（对或错），所以可以将布尔代数的所有操作用一个表来表示。在表2.1的第一行，我们列出基本操作，第二行中列出在 R 中的代码，表体中显示的是操作结果。

表 2.1: 布尔代数运算真值表

布尔代数表达式	A	B	not A	not B	A and B	A or B
R 中的表达式	A	B	!A	!B	A & B	A | B
	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE
	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE
	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE

习题

1. 从基本的布尔代数操作中能构建一些更复杂的操作。请写出 ``异或" 操作的真值表，并显示如何用 ``与"，``或"，``非" 来表达。
2. Venn 图可以用来说明集合的合并与相交。请用 Venn 图来画出对应的 ``与"，``或"，``非" 以及 ``异或" 操作。
3. DeMorgan 定理在 R 中的表达方式为 !(A & B) == (!A)|(!B) 和 !(A |B) == (!A) & (!B) ，请用语言表达这个公式，并用真值表来证明每个等式都是正确的。

2.4.2 R 中的逻辑操作

在 R 中有一类基本向量用来存储逻辑值。例如，一个基本的逻辑向量可以用这样的方式来构建：

```
> a <- c(TRUE, FALSE, FALSE, TRUE)
```

其输出是一个包含四个逻辑值的向量。逻辑向量可以用作索引：

```
> b <- c(13, 7, 8, 2)
> b[a]
```

```
[1] 13 2
```

在向量 b 中，与 $TRUE$ 相对应的元素被选择。如果试图对一个逻辑变量进行数学运算，例如：

```
> sum(a)
```

```
[1] 2
```

可以看出数学运算之前已经先将逻辑值 $FALSE$ 赋值为 0，把 $TRUE$ 赋值为 1，然后再做运算。本例的运算就是数一下在这个逻辑向量中有多少个值是 $TRUE$。

有两种方式可以用来表达布尔代数运算。通常使用 &，| 和 ! 这样的符号。他们都已经向量化了，可以看如下的例子：

```
> !a
```

```
[1] FALSE TRUE TRUE FALSE
```

如果对数字向量实施逻辑操作，那么数字 0 被当做 $FALSE$，所有非零值被当做 $TRUE$。

```
> a & (b - 2)
```

```
[1] TRUE FALSE FALSE FALSE
```

运算符 && 和 || 的作用与 & 和 | 相类似，不过有两点不同。第一，它们都不是向量化的操作，只是做一个计算。第二，它们确保从左到右进行运算，右边的操作符只是在必要的时候才执行。例如，如果 A 是 $FALSE$，那么不管 B 的值是什么，$A\&\&B$ 的值都将是 $FALSE$，所以对 B 不需要先进行计算。如果对 B 计算会消耗一定时间的话，那么采用这种运算就会节省时间，使计算变得更容易。这种方式有时候被称作短路计算。

习题

1. 在什么情况下，在计算 $A||B$ 时，需要对 B 进行计算?
2. 用上一节中的值，说出下面这些表达式的结果，然后在 R 中验证。

   min(b)

   min(a)

   max(b)

   max(a)

   length(a)
3. 写出 $b * a$ 的值。

2.4.3 关系运算符

在编程的时候，经常需要对真假关系进行检验。 R 允许对相等或不相等的关系进行进行检验，可以使用如下的运算符号 $>$， $<$， $==$, $>=$， $<=$， $!=$。

例如：构建一个向量：$a < -c(3, 6, 9)$

- 要检验哪个元素大于 4，可以输入：

  $a > 4$

- 要检验哪个元素等于 4，可以输入[3]：

  $a == 4$

- 要检验哪个元素大于等于 4，可以输入：

  $a >= 4$

- 要检验哪个元素不等于 4，可以输入：

  $a! = 4$

- 要输出大于 4 的元素，可以输入：

  $a[a > 4]$

- 构建向量 b

  $b < -c(4, 6, 8)$

- 要检验 a 中哪个元素大于 b 中的相应元素，可以输入：

  $a < b$

- 要输出 a 中小于 b 中相应元素的元素，可以输入：

  $a[a < b]$

2.5 数据输入与输出

在 R 中，允许从外部（如电脑硬盘上）读入数据，也允许将数据保存为外部文件。在讨论起应用之前，首先要知道这些数据从何处来，到何处去。

2.5.1 修改工作目录

如果您在 Windows 中使用 R , 那么可以使用鼠标从菜单栏上分别点击 File → Change dir 来指定数据读取和保存的目录。

用户也可以在命令窗口中键入 setwd() 命令来设置目录。例如，如果希望将目录指向位于 C 盘上的 mydata 目录，用户可以这样输入命令：

```
setwd("c:/mydata")   # 或 setwd("c:\\ mydata")
```

此后，所有默认的数据输入与输出目录就变成 $C:\mydata$ 了。[4]

[3] 做相等检验时要特别小心，因为 R 中存储的数字是近似值，所以您可能会得出 $49 * 4/49$ 不等于 4 的情况。

[4] 习惯使用 Windows 的用户可能觉得更改目录应该写成 $c:\mydata$，但是在 R 中，斜杠 (\) 是退出标志，已经被系统保留，如果用户想要使用斜杠的话，就需要键入两次，第一个 (\) 告诉系统第二个 (\) 不是一个退出符号。因为在有些系统中使用反斜杠 (/) 来指定路径，而且在 Windows 中使用两个斜杠很乏味，所以这两种方法 R 都接受。

2.5.2 DUMP() 和 SOURCE() 函数

假定用户已经创建了一个对象，名叫 (usefuldata)，为了以后能继续调用这个对象，希望能把它保存下来，这时可以输入：

```
dump("usefuldata", "useful.R")
```

这个命令在用户硬盘上创建一个文件 $useful.R$ 来保存向量 $usefuldata$。用户可以自己定义文件的名称，只要符合定义规则即可。

当用户想要重新使用该数据时，只要输入 $useful.R$ 来调用即可。这个命令执行后，在当前工作环境中创建 $usefuldata$ 这个对象。如果在此之前已经有了一个相同名字的对象，那么该对象就会被替换成新内容。

如果用户需要保存当前操作中的所有对象，可以使用：

```
dump(list=objects(), "all.R")
```

这个命令会在用户硬盘上创建一个文件，名叫 $all.R$，以后用户调用这个命令的时候，就能使用所保存的所有对象了。

如果希望将当前工作环境中的三个对象 $humidity$，$temp$ 和 $rain$ 保存到一个文件中，文件名为 $weather.R$，那么可以这样输入命令：

```
dump(c("humidity", "temp", "rain"), "weather.R")
```

习题

1. 使用文本编辑器，创建一个文件，包含这样一条命令
   $randomdata <- c(64, 38, 97, 88, 24, 14, 104, 83)$，然后把文件保存，文件名为 $randomdata$。
2. 在 R 中执行 $randomdata$ 文件，看看 $randomdata$ 向量是否被创建了。
3. 创建一个向量，名字叫 $numbers$，其中包含 (3 5 8 10 12) 这些数字。把这些数字存放到一个文件中，文件名为 $number.R$。用 rm() 命令删除当前工作环境中的数据，用 lm() 命令确认数据已经删除。最后再调用刚才保存的文件来重新导入向量数据。

2.5.3 重新指定 R 的输出

在默认方式下，R 会将其输出直接显示在屏幕上。用户也可以用 sink() 函数将输出保存为一个文件。例如，在 $greenhouse$ 文件中 $solar.radiation$ 数据，用命令 mean(solar.radiation) 会在屏幕上显示这个数据的均值。如果用户想把输出保存为一个文件，例如保存为 $solarmean.txt$，则可输入这样的命令:

```
sink("solarmean.txt") # 创建 solarmean.txt 作为输出文件
mean(solar.radiation) # 把均值写入 solarmean.txt
```

以后所有的输出都被保存到这个文本文件中，用户如果要取消输出到文件，可以用这样的命令：

```
sink() # 关闭 solarmean.txt; 将结果显示到屏幕上
```

此后，运算的输出重新显示在屏幕上。

2.5.4 保存及读入镜像文件

用户在运行 R 时所创建的所有向量和对象都被保存在一个称为全局环境的工作空间中。当退出 R 进程时，会提示用户是否需要保存工作空间。如果选择保存工作空间，则会在当前目录下创建一个名叫 $default.RData$ 的文件，这个文件中包含了当前工作空间中的所有信息。在 Windows 操作系统中，如果用户双击打开带有 $.RData$ 后缀的文件，R 会自动装载这个文件。如果这个文件没有保存在当前工作目录下，则可以用 load() 命令来加载。

当然用户也可以在没有退出 R 进程之前就保存当前工作空间。例如可以用下面的命令将当前工作空间保存为 $temp.RData$。

```
> save.image("temp.RData")
```

正如前面所说，可以通过双击直接打开 *temp.RData* ，也可以在打开 R 之后再将它加载到工作空间中。不过此时在当前工作空间中如果有相同的对象名，则会因为加载原有镜像文件而被替换。

2.5.5 数据框和 (READ.TABLE) 函数

一个数据文件通常包含有超过一列的数据，每一列代表一个变量。每一行通常表示一个观察值。例如，下面这个数据文件包含三个变量和四个观察值。

x	y	z
61	13	4
175	21	18
111	24	14
124	23	18

如果这样的数据保存在文件 *pretend.dat* 中，假定该文件在 *C* 盘下的 *myfiles* 目录中。我们就可以用这样的命令将数据读入到数据框中：

```
> pretend.df <- read.table("c:/myfiles/pretend.dat", header=T)
```

在数据框中，每一列都包含变量名，如果要查看 x 列的数据，可以用下列命令：

```
> pretend.df$x
```

2.5.6 数据列表

数据框事实上是一个特殊的列表，或可以称为结构。在 R 中列表包含所有的对象。用户不用自己来创建，有很多函数可以显示复杂的列表。用户可以用 names() 函数在列表中显示某个对象，也可以将其中的某一部分展开。

```
> names(d) # 列出对象名称
> d$x # 列出第 x 列的对象名称
```

函数 list() 可用于将多个片段的输出重新组织，例如：

```
> x <- c(3, 2, 3)
> y <- c(7, 7)
> list(x = x, y = y)
```

```
$x
[1] 3 2 3
$y
[1] 7 7
```

习题

1. 列出 *pretend.df* 中第一行第三列的元素。
2. 用两种不同的命令列出 *pretend.df* 中 y 列的元素。

2.6 本章习题

1. 将数据文件 $rnf6080.dat$ [5] 导入为数据框 $rain.df$ ，导入时使用 $header = FALSE$ 选项。

   (a) 列出 $rain.df$ 中第 2 行第 4 列的元素。

   (b) $rain.df$ 中每列的名称是什么？

   (c) 列出第二列中的内容。

   (d) 用下面的命令给数据框的每列重新指定标签。

   $>$ names(rain.df) <- c("year", "month", "day", seq(0, 23))

   (e) 创建一个新列，命名为 $daily$ ，其内容是 24 小时各列的和。

   (f) 作每天下雨总量的直方图。

2. 在区间 $[0, 6]$ 上绘制下列函数的图形。

$$f(x) = \begin{cases} 3x + 2, & x \leq 3 \\ 2x - 0.5x^2, & x > 3 \end{cases}$$

[5] 该数据可以从以下网站获得 www.stats.uwo.ca/faculty/braun/data/rnf6080.dat

第 三 章

绘制统计图形

在统计计算过程中，通常需要绘制一些关于数据的图表，一些计算结果也常常要用图形来显示。在这一章中，首先概述一下这一过程在 R 中如何实现，然后学习一些基本的绘图方法。接着，会介绍一下怎么选择绘图的类型，这个过程通常并不简单，有很多反面例子供我们学习。最后，会学习如何绘制自定义图形。

在 R 中，有好几个不同的图形系统。最早的那一个通常用来和 S 中的图形系统做比较，现在也被称为基本图形系统。我们可以把这个基本图形系统想象成用墨水在纸上绘图，你用某种固定的东西在画图，而一旦在上面画了一笔，它就永远存在在那里。虽然你还可以再用其他的东西来覆盖它，或者把它移到另一张干净的纸上，但它永远存在那里。基本的图形系统，在它设计之初，就允许很容易地创建高质量的科学绘图。在这一章中，主要集中在这个基本绘图系统上。

网格 (grid) 程序包提供了新一代图形系统的基础。它同样能创建高质量的图形，但程序员可以控制图形的每一个部分，图形看起来更像一个物理模型，用户创建它并显示它，而不仅仅是把它画出来。*lattice* 和 *ggplot* 这两个程序包提供了基于网格图形的高级绘图功能。

基本图形系统和网格图形系统都被设计成``与设备无关"。用户用命令指示从何处、向哪个方向绘图，并且这些命令在不同的设备上都能工作。图形的实际显示会因设备的不同而稍有变化，例如显示在纸上和显示在屏幕上会稍有不同，这是因为两者功能不同。

在 R 中还有多个外部图形系统，它们可提供交互式图形，三维显示等，不过这些超出了本书的范围。

3.1 高级绘图

在这一节中，我们将讨论几个基本的绘图功能。这些功能在 R 中被称为``高级" 功能，因为用户不用担心画笔走向的细节，用户只需要描述自己的需求，然后 R 就会完成剩余的工作。

3.1.1 柱状图和点状图

统计图形最基本的功能是描述数据集。柱状图和点状图分别用柱的长度和点的位置来表示相应的数字。

例如 R 中自带的数据文件 $VADeaths$，它是 1940 年 Virginia 州的人口统计数据，其中包含了每 1000 人中每年的死亡人数。这个数据可以用柱状图来表示。

```
VADeaths
```

	Rural Male	Rural Female	Urban Male	Urban Female
50-54	11.7	8.7	15.4	8.4
55-59	18.1	11.7	24.3	13.6
60-64	26.9	20.3	37.0	19.3
65-69	41.0	30.9	54.6	35.1
70-74	66.0	54.3	71.1	50.0

```
barplot(VADeaths, beside=TRUE, legend=TRUE, ylim=c(0,90),
+ ylab="Deaths per 1000",
+ main="Death rates in Virginia")
```

图 (3.1) 显示了绘图的结果，每个柱代表矩阵中的数。其中选项 $beside = TRUE$ 表示每一列中的数值依次显示。$legend = TRUE$ 选项表示在图的右上方显示图例。$ylim = c(0, 90)$ 的作用是设定柱在垂直方向的高度，这样就有足够的空间来放置图列。我们将在第 3.3 节中对如何设置图例做更多的介绍。最后 $main = argument$ 选项的作用是在图上方显示标题。

另一个能实现相同功能的绘图方法是作点状图 (如图3.2)

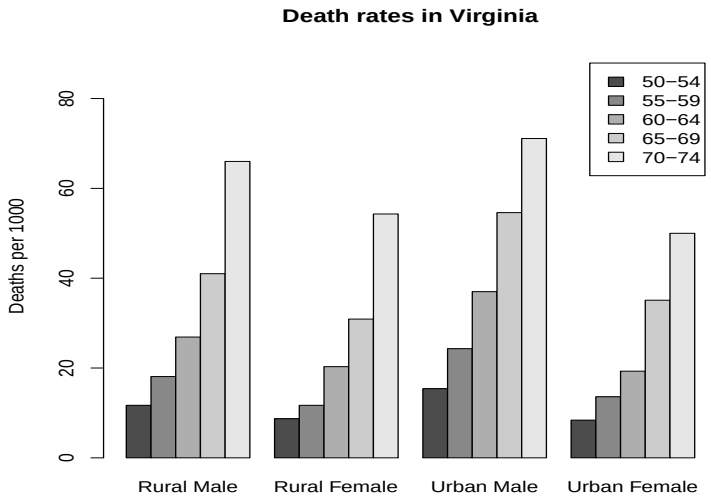


图 3.1: 一个柱状图例子

```
dotchart(VADeaths, xlim=c(0, 75),
+ xlab="Deaths per 1000",
+ main="Death rates in Virginia")
```

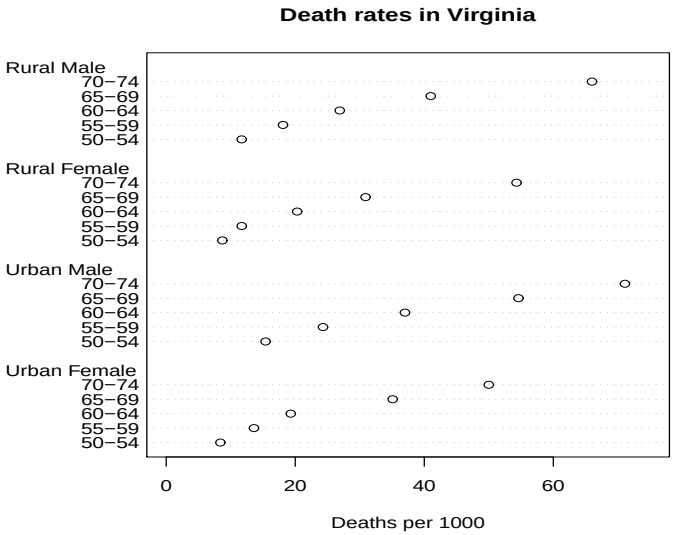


图 3.2: 一个点状图例子

我们把 x 轴的区间设定为从 0 到 75 之间，把 0 包含在内的目的，是可以在不同的组之间比较总的比例。

3.1.2 饼图

饼图就是把一个圆盘按所需表达变量的观察数划分为若干份，每一份的角度（即面积）等价于每个观察值的大小。例如，图 (3.3) 中的字母表达的就是每组的数量大小，它是用下面的代码生成的。

```
groupsizes <- c(18, 30, 32, 10, 10)
labels <- c("A", "B", "C", "D", "F")
pie(groupsizes, labels,
+ col=c("grey40", "white", "grey", "black", "grey90"))
```

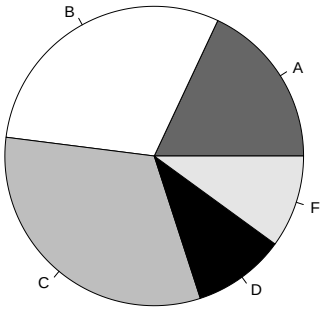


图 3.3: 一个饼图例子

饼图在一些非科技出版物中使用非常广泛，不过统计学家使得的很少，我们在 3.2 节中讨论其中的原因。

3.1.3 直方图

直方图是一种特殊的柱状图，它用来表示观察值的频率分布。每个柱子代表在某个区间内变量 x 的数值。不同之处在于，在 R 中默认每个柱子的宽度都一样。在这个例子中，每个柱子的高度代表在相应区间内的观察值数量。如果柱子的宽度不一样，那么柱子的面积代表相应的数量。此时，高度代表密度，即每单位 x 的频率。

在 R 中，绘制直方图的函数是 hist(x,...)。其中 x 是一个向量，它包含了一定数量的观察值。在命令中可以添加若干选项来控制输出。例如，图 (3.4) 就是用下列程序生成的。

```
x <- rnorm(100)
hist(x)
```

如果变量 x 有 n 个观察值，那么 R 默认把 x 的区间大约按 $\log_2(n)+1$ 的间隔来划分，以此决定柱子的数量。例如上面的数据包含 100 个观察值，因为：

$100 > 2^6 = 64$

$100 < 2^7 = 128$

$6 < log2(100) < 7$

所以 R 会选择 7 到 8 个柱。而事实上它选择了 11 个柱子，因为它同时也试图将分割点放在整数点上（在这个例子中是乘以 0.5 。）

上面这个规则（被称作 Sturges 规则）当面对很大的 n 时可能不再符合，会给出过少的柱。目前有研究认为合适的柱子数应该是 $n^{1/3}$ 而不是 $\log_2(n)+1$ 。选项 $breaks = "Scott"$ 和 $breaks = "Freedman-Diaconis"$ 可用作这方面的选择。例如图3.5表示了用这两种规则（ Sturges 和 Scott ）显示有 10000 个观察值时的差别。

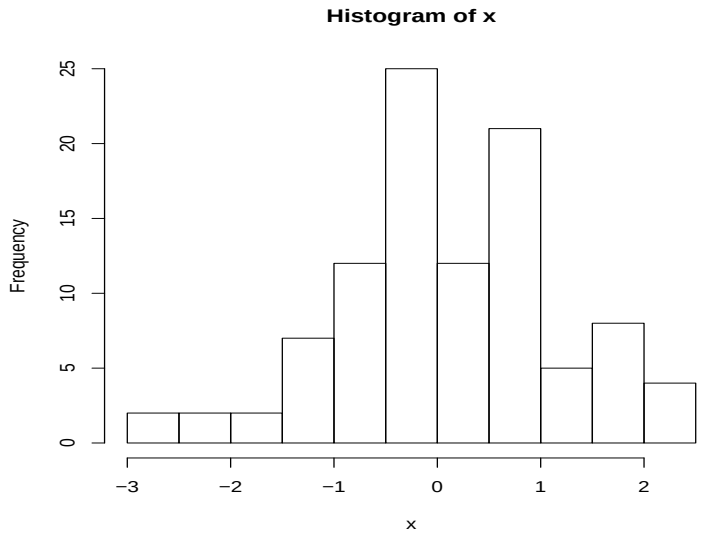


图 3.4: 100 个正态分布随机数的直方图

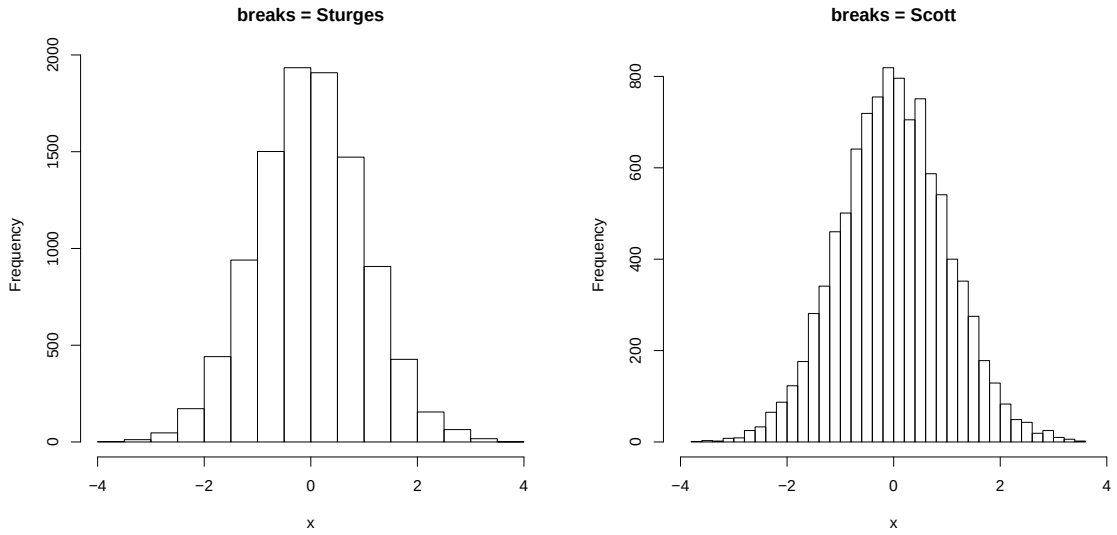


图 3.5: 两种方式得到的直方图

3.1.4 箱线图

箱线图（Boxplot）也称箱须图（Box-whisker Plot），是利用数据中的五个统计量：最小值、第一四分位数、中位数、第三四分位数与最大值来描述数据的一种方法，它可以粗略地看出数据是否具有对称性以及分布的分散程度等信息，特别可以用于对几个样本的比较。

盒子代表数据的中间部分，两边的线段（须）代表了大部分的数据范围。在有些情况下，异常值（即与数据集中的其余部分有很大差异的观察点）通常被绘制成独立的点。

绘制箱线图的基本结构如下：

1. 在中位数的位置画一条水平线。
2. 画一个矩形盒，上下两边的位置分别对应数据的上下四分位数（Q1 和 Q3）。
3. 在 $Q3 + 1.5IQR$（四分位距）和 $Q1 - 1.5IQR$ 处画两条与中位线一样的线段，这两条线段为异常值截断点，称为内限。处于内限以外位置的点表示的数据都是异常值。
4. 从矩形盒两端边向外各画一条线段直到不是异常值的最远点，表示该批数据正常值的分布区间。

这样就用矩形盒定义了四分位数的范围 (IQR)。这是上四分位和下四分位的区别。因为有 50% 的数据会落在盒子内，所以我们用 IQR 来度量数据中央部分变化的总量。下面的须从盒子的底部到最小值不低于 1.5 倍四分位数的地方。同样，上面的须也是从盒子的顶部到最大值不大于 1.5 倍四分位数的地方。之所以采用这样的定义，是因为如果数据是正态分布或其他类似分布的话，大约 99% 观察将介于盒须之间。

图3.6 是一个箱线图说明例子。箱线图可以很方便地用于比较两个数据的分布，每一个数据包含 10 个或以上的观察值。例如，在 R 中自带的 *iris* 数据集就是一个被很好研究过的数据，它包含三种鸢尾花的各 50 个数据。图3.7 就是由下面的代码生成。

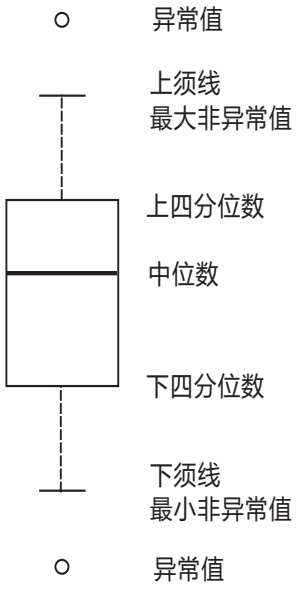


图 3.6: 箱线图例子

```
boxplot(Sepal.Length ~ Species, data = iris,
+   ylab = "Sepal length (cm)", main = "Iris measurements",
+   boxwex = 0.5)
```

从图中可比较不同花种萼片长度的分布。这里使用了 R 的公式来产生图形函数。其中 *Sepal.Length Species* 表示依赖于花种的萼片长度。这两个变量都是数据文件 *iris* 中的一列。然后 boxplot() 函数将每一花种都绘制一幅箱线图，并将它们并排排列。从图中可以看到每种花萼片长度的均值有明显的差别，而在 *virginica* 样本中，则有一个标本的值非常小，显得很不寻常。

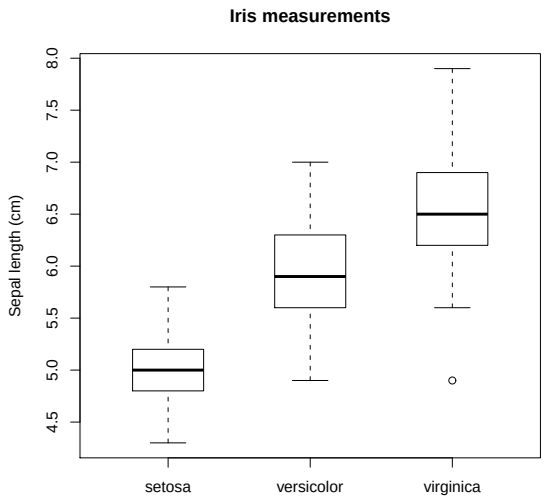


图 3.7: 多个箱线图排列例子

3.1.5 散点图

在做统计研究时，人们最关心变量之间的关系。要做这样的研究，一个最常用的方法是绘制散点图，由两个变量 x_i 和 y_i 确定点的位置并绘制在图上，这样就可以显示出 x_i 和 y_i 的关系。在 R 中，散点图（以及其他很多类似的图）是使用函数 plot() 来绘制的。它最基本的命令是 $plot(x, y, \dots)$，其中 x 和 y 是长度相等的数值向量。这个绘图函数还有很多选项可用，也有可用于绘制非数值数据的函数。

一个重要的选项是图的类型，默认的类型是 $type = "p"$，即绘出散点图。如果用 $type = "l"$ 的话，绘出来的就是折线图，即用一段一段的折线分别连结 (x_i, y_i) 点。还有其他很多类型，比如 $type = "n"$ 就是绘制一张空图，即绘制一个框架。更详细的内容，我们将在 3.3 节中说明。

用这个函数可以绘制很多种不同的图像，下面用例子来作演示。首先生成两列数据，其中一列是标准正态分布，另一列是均值为 30 的 *Poisson* 分布。

```
x <- rnorm(100)       # 生成 100 个正态分布随机数
y <- rpois(100, 30)   # 生成 100 个均值为 30 的泊松分布随机数
mean(y)               # 它的均值应该在 30 附近
```

```
[1] 30.91
```

```
plot(x, y, main = "Poisson␣versus␣Normal")
```

其中选项 *main* 是给图片设一个主标题。其结果如图 (3.8) 所示。

读者还可以尝试一下其他选项的效果，比如：

```
plot(x, y, type="l") # 绘制折线图
plot(sort(x), sort(y), type="l") # 绘制分位数
```

要获取 plot() 函数其他选项的使用方法，可以使用 ?*plot* 或 ?*par* 查看帮助文件。

3.1.6 QQ 图

分位数 -分位数图（通常缩写成 QQ 图）是一种典型的散点图，它可用于比较两个数组的分布，或者将一个样本和某个参考分布做比较。

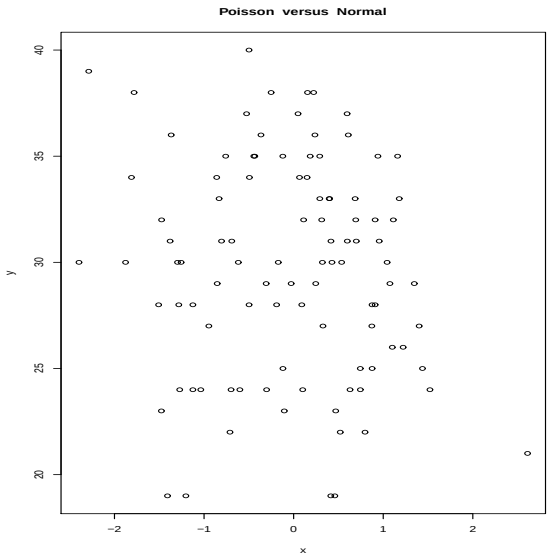


图 3.8: 一个散点图例子

当两个样本的大小相等，那么绘制 QQ 图时首先将两组数据排序，即 $X[1] \leq \cdots \leq X[n]$ 和 $Y[1] \leq \cdots \leq Y[n]$ 都从小到大排序，然后绘制这两者的散点图。

当两组数据的大小不相等的时候，就需要用某些技巧来使两者相吻合。在 R 中使用的方法是，将较大的那组数据缩减来匹配较小的数据，在此过程中保持最大值和最小值不变，在最大值和最小值之间选择相等的分位数间距。假设有两组数，其中 x 有 5 个值， y 有 20 个值，那么 x 中的值就应该对应 y 中的最小值，下分位数，中值，上分位数和 y 的最大值。

假如要将一个样本对应于某个参考分布来绘图，那么就是用理论的分位数值作为对应的参考部分。在 R 中通常将理论值放置在 x 轴上，将样本数据放置在 y 轴上，不过也有些作者倒过来设置，读者需要注意。为了避免偏差，可依据 $(i-1/2)/n$ 的概率来确定分位数，这样就能在 0 到 1 之间平均分布。

当 x 和 y 的分布相吻合时，QQ 图中的点就应该位于 $y = x$ 的直线上。假如其中一个分布是另一个分布的线性变化，那么在 QQ 图中就得到不同的直线。另一方面，如果两个分布不匹配， QQ 图会显示一些系统图案。下面的代码绘能制出一些典型的图案（参阅图:3.9）

```
X <- rnorm(1000)
A <- rnorm(1000)
qqplot(X, A, main="A and X are the same")
B <- rnorm(1000, mean=3, sd=2)
qqplot(X, B, main="B is rescaled X")
C <- rt(1000, df=2)
qqplot(X, C, main="C has heavier tails")
D <- exp(rnorm(1000))
qqplot(X, D, main="D is skewed to the right")
```

习题

1. 在 islands 向量中包含了 48 个岛屿的数据。
   - 绘制这些数据的直方图。
   - 在绘直方图之前，对面积数据取对数有优势么？
   - 在取对数与不取对数的情况下，分别比较用 Struges 和 Scott 法所绘制直方图的差异。
   - 分别在取对数和不取对数情况下，绘制这些数据的箱线图。
   - 绘制面积数据的点状图，在这里采用对数变化有必要吗？

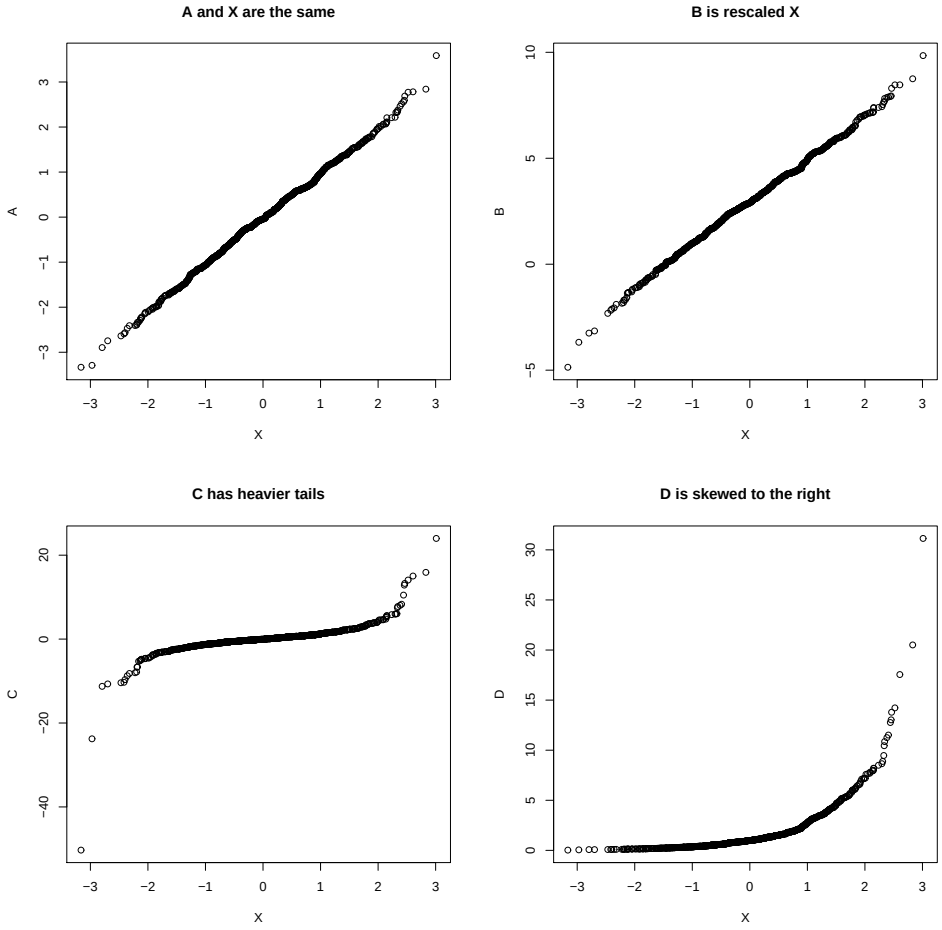

A and X are the same
B is rescaled X
C has heavier tails
D is skewed to the right
A
B
C
D
X

图 3.9: 几个 QQ 样图

- 您认为对于这个数据，采用什么图来描述最合适？

2. stackloss 数据来自工厂中用氨气生产硝酸的过程，包含 4 个变量和 21 个观察值。前面三个变量分别是 Air.Flow， Water.Temp 和 Acid.Conc 。第四个变量是 stack.loss ，它表示氨气在被吸收之前泄露的值（欲知详情，可参阅该数据的帮助文件）。

   - 使用散点图来探索一下酸度，水温，气流和氨气泄露值之间可能的相关关系。它们是线性的关系吗？
   - 使用 $pairs()$ 函数绘制四个变量的两两散点图。看看哪些变量之间有线性关系，哪些不是线性关系。

3. 在压力数据中包含两个温度和压力变量。

   - 用 y 轴代表压力，用 x 轴代表温度画散点图，它们是线性关系吗？
   - 下面这个函数的图能很好地穿过刚才的散点图：$y = (0.168 + 0.007x)^{20/3}$ 由曲线所代表的压力值和观察值之间的差称为残差，可以用下面的命令来计算这个残差值。

```
residuals <- with(pressure,pressure-
+ (0.168+0.007*temperature)^(20/3))
```

     绘制这个残差值的 QQ 图，检查一下它们是不是服从正态分布，分布是否有偏。
   - 用 $y^{3/20}$ 将压力数据转换一下，然后重新对温度变量画散点图。现在两者之间是线性关系了吗？
   - 重新计算残差值，现在这个残差值有正态分布了吗？

3.2 选择合适的图形

我们已经学习了柱状图，点状图，饼图，直方图，箱线图，散点图和 QQ 图。还有很多其他统计图型没有在这里讨论，那么应该选择哪种图来描述统计数据呢？

首先要判断数据的类型。如前面所述，柱状图，点状图和饼图用于描述单个的值，直方图，箱线图和 QQ 图用来描述一个分布，散点图则用来描述两个变量之间的关系。

其次要考虑面向的对象。如果所绘制的图是给自己看或者给具有统计学知识的读者看，那么可以假定他们有较高的理解能力。例如箱线图和 QQ 图就需要更多的解释才能理解，而直方图则相对简单一点，故前者不宜用于面对普通读者。

最后还需要对人的视觉感知有一定的了解，这方面已经有了很多的研究，我们在此只涉及其中很小的一部分。当人们看一张图时，视觉系统从图中分解出一些数量信息，这个过程可以描述为对长度，位置，斜率，角度，面积，体积以及各种颜色的感知。研究发现，人们感知长度和位置的能力比感知斜率和角度的能力强，而对于面积和体积的感知则很不精确。大部分人对颜色差异的感知能力都较高，不过大约有 10% 的男人以及很少一部分的女人是色盲，几乎没有人擅长于用颜色来做出度量。

在绘制图形时要考虑上面这些因素，所以要尽量用人们容易感知的方式来表达重要的信息，尽量避免在一个图中包含矛盾的信息。

例如，在柱状图中的各个柱子就很容易被识别，因为柱的位置和长度可以看得很清楚，柱子的面积还能强化我们的感知。

不过柱状图柱子的长度和面积约束了我们，所以通常将柱的底设为零，这样它们的位置，长度和面积能传达同样的信息。如果要描述与零值不相关的数字，那么点状图就是较好的选择，在点状图中我们主要感知点的位置。

根据视觉原理，饼图并不是一个描述数据的好选择。为了感知饼图中每个切片的尺寸，人们需要度量角度和面积，而这些并不是他们所擅长的。

用颜色用来区分不同组别的数据是一个很好的选择。在 RColorBrewer 软件包中包含了多个调色板，可用于选择各种颜色。有些调色板用于表达从低到高的序列，有些表达与中值的偏离度，还有些纯粹表达数值。选择好的调色板能保证大部分人（即使是色盲）也能很好地感知颜色的差异。

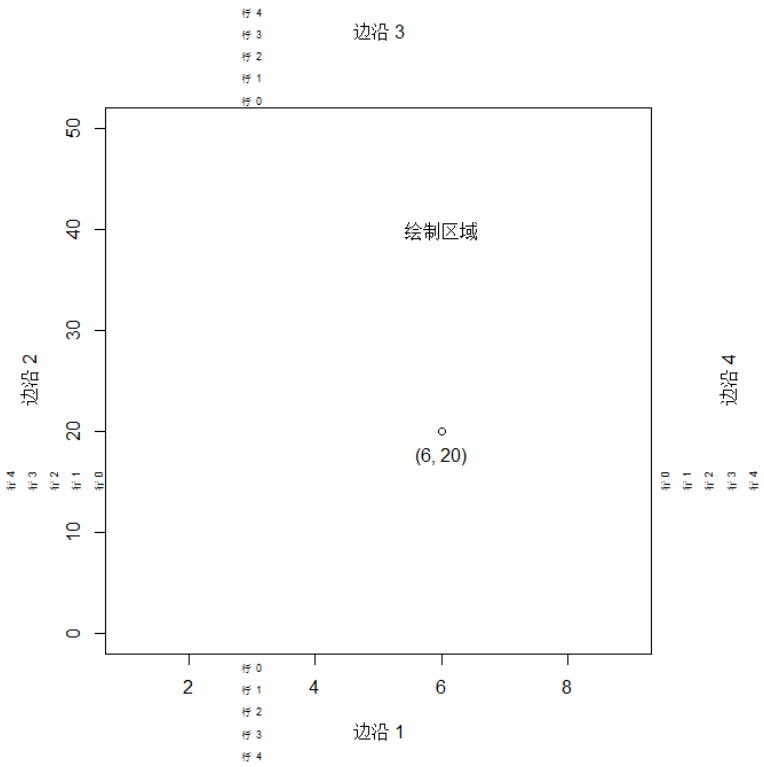


图 3.10: R 绘图的内部结构

3.3 低层绘图函数

类似于 barplot()， dotchart() 和 plot() 这样的函数采用低层的绘图函数来画线和点，来表达它们在页面上放置的位置以及其他各种特征。

在这一节中，我们会描述一些低层的绘图函数，用户也可以调用这些函数用于绘图。首先我们先讲一下 R 怎么描述一个页面；然后我们讲怎么在页面上添加点，线和文字；最后讲一下怎么修改一些基本的图形。

3.3.1 绘图区域与边界

R 在绘图时，将显示区域划分为几个部分。绘制区域显示了根据数据描绘出来的图像，在此区域内 R 根据数据选择一个坐标系，通过显示出来的坐标轴可以看到 R 使用的坐标系。在绘制区域之外是边沿区，从底部开始按顺时针方向分别用数字 1 到 4 表示。文字和标签通常显示在边沿区域内，按照从内到外的行数先后显示。图 (3.10) 显示了绘图的结构。图中可见坐标轴标记在第一行，而我们标记的“边沿”绘制在第三行。

3.3.2 添加对象

在绘制的图像上还可以继续添加若干对象，下面是几个有用的函数，以及对其功能的说明。

- points(x, y, ...)，添加点
- lines(x, y, ...)，添加线段
- text(x, y, labels, ...)，添加文字
- abline(a, b, ...)，添加直线 $y = a + bx$
- abline(h=y, ...)，添加水平线

- abline(v=x, ...)，添加垂直线
- polygon(x, y, ...)，添加一个闭合的多边形
- segments(x0, y0, x1, y1, ...)，画线段
- arrows(x0, y0, x1, y1, ...)，画箭头
- symbols(x, y, ...)，添加各种符号
- legend(x, y, legend, ...)，添加图列说明

上面这些函数中的其他可选参数包括颜色、大小，以及其他属性。
例如成年人食指长度和宽度的数据，数据名为 indexfinger，其结构如下：

	sex	length	width
1	M	7.9	2.3
2	F	6.5	1.7
3	M	8.4	2.6
4	F	5.5	1.7
5	F	6.5	1.9
6	M	8.0	2.1
7	F	7.0	1.8
8	M	7.5	1.9

通过散点图可以大致显示长度和宽度之间的关系

```
plot(width ~ length, data=indexfinger)
```

假设在刚才绘制的图上，希望标记出男女手指最长的点（即第三和第七个观察值），那么可以用 points() 函数来添加。

```
with(indexfinger[c(3,7),], points(length, width, pch=16))
```

修改之后的结果显示在图 3.11(a) 中。上面的命令中，参数 $pch = 16$ 是将默认的空心点改成实心点（其他值会显示别的图形），甚至可以用不同的文字来表示，比如 $pch = "f"$ 会用字符 f 来画这两个点。
其实在 plot() 函数中就可以设定 *pch* 的值，例如：

```
plot(width ~ length, pch=as.character(sex), data=indexfinger)
```

随后可以分别对男性和女性的数据进行线性回归，并添加图列（见图3.11(b)）。

```
abline(lm(width~length, data=indexfinger, subset=sex=="M"), lty=1)
abline(lm(width~length, data=indexfinger, subset=sex=="F"), lty=2)
legend("topleft", legend=c("Male", "Female"), lty=1:2)
```

读者可能还希望在绘图区域之外添加一些注释，下面这些函数包含了这些功能：

- title(main, sub, xlab, ylab, ...)，添加主标题，副标题，坐标轴标签等。
- mtext(text, side, line, ...)，在边沿处添加文字。
- axis(side, at, labels, ...)，在图中添加坐标轴。
- box(...)，在绘制区域外显示方框。

例如，图 (3.10) 就是用下面的代码来绘制的：

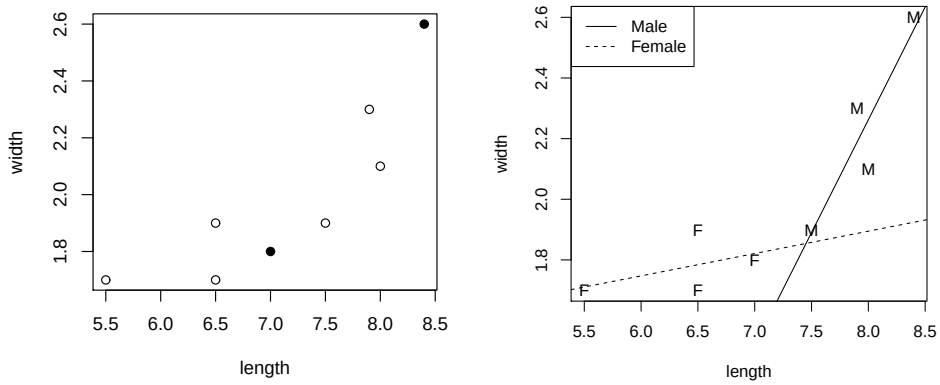


图 3.11: 对图进行修改的例子

```
par(mar=c(5, 5, 5, 5) + 0.1)
plot(c(1, 9), c(0, 50), type="n", xlab="", ylab="")
text(6, 40, "绘制区域")
points(6, 20)
text(6, 20, "(6,␣20)", adj=c(0.5, 2))
mtext(paste("边沿", 1:4), side=1:4, line=3)
mtext(paste("行", 0:4), side=1, line=0:4, at=3, cex=0.6)
mtext(paste("行", 0:4), side=2, line=0:4, at=15, cex=0.6)
mtext(paste("行", 0:4), side=3, line=0:4, at=3, cex=0.6)
mtext(paste("行", 0:4), side=4, line=0:4, at=15, cex=0.6)
```

我们将在下面一节中说明 par() 这个函数。

3.3.3 设置图像参数

当绘一幅新图时，有两个选项可以设置这幅图的总参数。首先是何时打开绘图设备。 R 通常会默认自动打开一个窗口设备，但用户也可以明确指定打开何种绘图设备，并且在需要的时候将其打开。有些命令根据操作系统各不相同，常用的一些绘图设备包括：

- windows(...)，在 MS Windows 中打开一个屏幕窗口。
- x11(...)or X11(...)，在 Unix 类系统中打开一个屏幕窗口。
- quartz(...)，在 Mac OSX 中打开一个屏幕窗口。
- postscript(...)，打开一个文件用于存放 Postscript 图形输出。
- pdf(...)，打开一个文件用于存放 PDF 图形输出。
- jpeg(...)，打开一个文件用于存放 JPEG 图形输出。
- png(...)，打开一个文件用于存放 PNG 图形输出。

读者可以参阅对应的帮助文件，里面有各函数详细的参数说明，包括图形大小，背景颜色等。

在绘图设备打开之后，图形的其他参数就可以用 par() 函数来设定。这个函数有大量的参数，我们在此只列出其中的一小部分，其他可参阅帮助文件。

- mfrow=c(m, n)，告诉 R 在同一页面总要绘制 m 行 n 列个图形，而不是每个图新开一个页面。

- mfg=c(i, j)，告诉 R 在第 i 行，第 j 列绘图。
- ask=TRUE，告诉 R 如果绘制新图会擦除原有图时，要提醒用户。
- cex=1.5，告诉 R 在绘制区域扩展字符对应的量。还有其他单独的参数，如 cex.axis，来控制边沿区域的文字。
- mar=c(side1, side2, side3, side4) 在每一方向上按给定的数字设定绘图区域的边沿。
- oma=c(side1, side2, side3, side4) 设置外边沿。
- usr=c(x1, x2, y1, y2) 按照给定的 x 和 y 的区间，在绘图区域内设定坐标系。

可以用多种方法来设定 par() 函数的参数，如果用字符串（比如 par("mfrow")），那么函数返回图形参数的当前值。如果用指定的参数（比如 par(mfrow=c(1,2)），那就设定成了相应的参数。最后用户可以用 list 命令来一次性输入多个参数。

3.4 本章习题

1. 使用 islands 数据，使用下面的代码绘图。

```
hist(log(islands,10), breaks="Scott", axes=FALSE, xlab="area",
+ main="Histogram␣of␣Island␣Areas")
axis(1, at=1:5, labels^=10(1:5))
axis(2)
box()
```

   (a) 解释这些代码每一步代表什么意思。
   (b) 给图像添加一个副标题: " 根据 10 为底的对数作图"。
   (c) 修改代码，使用"Struges" 方法来绘图，使用 round() 来确保坐标轴标签上没有多余的数字。

2. 使用 pressure 数据作图。
   (a) 先绘制 pressure 和 temperature 关系图，然后用下面的代码添加曲线

```
curve((0.168 + 0.007*x)^(20/3), from=0, to=400, add=TRUE)
```

   (b) 按 $y^{3/20}$ 转换 pressure 数据，然后再次画图。现在是线性还是非线性关系？使用 abline() 函数添加一直线（需要设定截距和斜率）
   (c) 添加一个副标题。
   (d) 使用 mfrow() 将上面的过程绘制在 2×1 的格式输出。然后再按 1×2 的格式输出。

第 四 章

R 语言编程

编程涉及到编写较为复杂的指令系统，主要有两种类型：命令式编程（例如 R 中使用的方式），把一连串指令结合在一起，告诉电脑做什么；声明式编程（例如在 HTML 网页文件中使用的方式），只是声明程序的最终结果，不提供有关如何到达这种结果的细节。在每个类型中，还包括多个分支，一个特定的程序可能涉及其中的几个方面。例如， R 执行的可能是一段应用程序（描述需要通过哪些步骤能实现某一任务），也可以是一个模块（分解为自我包含的软件包），还可能是面向对象的操作（即被组织起来对复杂对象的描述性操作），当然也可以是一个特定功能函数（将程序组织为一个函数集合，它能实现某个特定的计算而不会有外部效应）。本书主要关注如何编写应用程序。

正如第一章中所描述， R 语句主要包含需要被执行的表达式。大多数程序都需要执行重复的操作，但重复的数量取决于用户的输入。这一章首先阐述几个流程控制（flow control）命令，它们能控制某些命令并重复执行多次。本章的其余部分就如何设计和调试程序提出一些建议。

4.1 流程控制

4.1.1 FOR() 循环语句

本书的目的之一是介绍如何进行随机过程仿真。仿真通常需要执行大量的重复过程，因为我们想得到的是行为的模式，而不仅仅是一个简单的例子。

$for()$ 语句的功能是让某个操作重复执行特定的次数。

该语句的语法为：for (name in vector) {commands}

这段语句设定了一个称为 *name* 的变量，并将它等于向量中的每一个元素。对于每一个值，大括号内的命令将被执行。大括号的作用是将若干命令合并成一组，这样 R 将它们作为同一个命令来对待。如果只有一个执行命令，那么大括号也可以省略。

例 4.1

斐波那契数列（Fibonacci sequence）是一个著名的数学序列，在这个数列中，第一第二个元素被定义为 1 ，随后的元素被定义为前面两个元素的和。例如第三个元素就是 2 ，它等于前两个元素的和，即 $2 = 1 + 1$ 。第四个元素是 $3(= 1 + 2)$ ，第五个元素是 $5(= 2 + 3)$ ，后面的以此类推。

如果用 R 来产生前面 12 个元素，可以使用这样的语句：

```
> Fibonacci <- numeric(12)
> Fibonacci[1] <- Fibonacci[2] <- 1
> for (i in 3:12) Fibonacci[i] <- Fibonacci[i-2]+Fibonacci[i-1]
```

这个程序中，第一句定义一个变量，名叫 Fibonacci ，它的属性是长度为 12 的一个数量向量，这个向量的初始值包含了 12 个 0 。第二行命令将向量的第一和第二个元素赋值为 1 。第三行命令按斐波那契数列构建的原则分别给各元素赋值。例如第三个元素是第一第二个元素的和，$Fibonacci[3] = Fibonacci[1] + Fibonacci[2]$ ，第四个元素赋值为第二第三个元素的和， $Fibonacci[4] = Fibonacci[2] + Fibonacci[3]$ ，所以他们分别为 2 和 3 。用 for() 循环语句就可以计算出这个数列的第三到第十二个元素的值。

需要显示这十二个数值的话，可以直接输入变量名：

```
> Fibonacci
```

```
[1] 1 1 2 3 5 8 13 21 34 55 89 144
```

例 4.2

假设车行出售一辆价值两万美元的新车，顾客可以有两种付款方式。第一种方式是，付全款，然后会得到一千美元的折扣。第二种方式是，顾客分二十个月分期付款，每月支付一千美元，但顾客不需要支付利息。

事实上采用第一种付款方式时，该车实际的销售价格是 $19000，而在第二种付款方式中，车行实际上收了一定的利息。我们可以用下面的公式来计算这个利息率 i：

$$19000 = 1000\left(\frac{1-(1+i)^{-20}}{i}\right).$$

两边都乘以 i 并除以 19000 之后，得到这样一个*不动点*问题的形式:

$$i = \frac{\left(1-(1+i)^{-20}\right)}{19}.$$

给 i 设定一个初始估计值，并把它代入到方程的右侧，通过计算后就可以得到一个新的 i 值。假如初始的估计值是 $i = 0.006$，那么新的 i 值就是：

$$i = \frac{\left(1-(1+0.006)^{-20}\right)}{19} = 0.00593.$$

把这个计算得到的 i 再次代入到方程右侧，又可以得到一个新的 i 值：

$$i = \frac{\left(1-(1+0.00593)^{-20}\right)}{19} = 0.00586.$$

这种不动点问题，通常需要经过多次迭代才能得到一个确定的解。下面用 R 语句进行 1000 次迭代来解这个问题：

```
> i <- 0.006
> for (j in 1:1000) { i <- (1-(1+i)^(-20))/19 }
> i
```

```
[1] 0.004 935 593
```

习题

1. 按下列要求编写程序来产生斐波那契数列：
   a. 将第一第二个元素都设为 2。
   b. 将第一第二个元素分别设为 3 和 2。
   c. 把产生序列的算法从加改为减，例如第三个元素的值等于第二个元素减去第一个元素，以此类推。
   d. 将产生序列的算法修改为前三个元素的和，为此将第三个元素的值设为 1。

2. 设 f_n 是第 n 个斐波那契数。

   a. 构建一个数列，它等于前后两个元素的比，即 $f_n/f_{n-1}, n = 1, 2, \cdots, 30$ 。这个数列能收敛吗？

   b. 已知黄金分割率等于 $(1+\sqrt{5})/2$ ，上述数列收敛到这个黄金分割率吗？能否给出证明。

3. 给出下面各个语句的值，并在 R 中进行验证。

   a. > answer <- 0

   > for (j in 1:5) answer <- answer + j

   b. > answer <- NULL

   > for (j in 1:5) answer <- c(answer, j)

   c. > answer <- 0

   > for (j in 1:5) answer <- c(answer, j)

   d. > answer <- 1

   > for (j in 1:5) answer <- answer * j

   e. > answer <- 3

   > for (j in 1:15) answer <- c(answer, (7 * answer[j]) %% 31)

   检查这个数列的最后结果。如果你不知道产生这个序列的规则，你还可以预测后续元素吗？

4. 参考本节中车行卖车的例子，分别计算初始估计利率 $i = 0.006$ ， $i = 0.005$ 和 $i = 0.004$ 时，前面 20 个递归值。根据这三次计算的结果，真实的利率应该是多少（精确到小数点后 5 位）？

5. 用不动点递归的方法计算在区间 $[0, 1]$ 中，方程 $x = \cos(x)$ 的值。初始值使用 0.5 时，需要迭代多少次才能使最后结果的前两位数字一致。又要迭代多少次才能使结果的前三位，前四位数字一致？如果把初始值修改为 0.7 和 0.0 的话结果又是如何呢？

6. 重复上面那个问题，只是方程修改为 $x = 1.5\cos(x)$ ，（答案应该接近于 $x = 0.9148565$ ）。

   a 不动点递归方法能收敛吗？如果不收敛，把方程修改为 $x = \cos(x)/30 + 44x/45$ 。现在递归能收敛吗？

   b 你能证明这两个方程的解相同么？

   c 计算 $1.5\cos(x)$ 和 $\cos(x)/30 + 44x/45$ 的导数，有理论说，如果这个值小于 1 时，当初始值接近于真实解的时候，不动点递归就能收敛。这能解释问题 (a) 中的现象吗？

4.1.2 IF() 语句

前面学习了逻辑向量和关系运算符。下面介绍的语句有一个很强大的功能，能在不同的场合进行不同的操作。 if() 语句的作用就是让用户选择执行某些命令，使得程序执行更方便。

该命令的语法为：

if(条件) {如果为真时调用的命令 }

if(条件) {如果为真时调用的命令 } else {条件为否时调用的命令}

当条件为真时，可以调用一系列命令。语句中 else 部分是可选的，它给出了条件不为真时可选择的执行命令。不过不要将它们写成两行，例如：

if(条件) {如果为真时调用的命令 }

else {条件为否时调用的命令}

这样会给出出错信息，因为当你还没来得及输入第二行时， R 就会首先执行第一行命令。如果这两行命令包含在一对大括号之内，那么就不会出错，因为 R 会首先读完所有的命令，然后才执行它，例如：

if(条件) {
如果为真时调用的命令
} else {
条件为否时调用的命令
}

在 R 中同样也允许使用数值作为判断条件，其中零值被看做``否"，其他任何非零值都被看做``真"，不过缺失值会导致出错。

例 4.3

一个简单的 if() 语句例子：

```
> x <- 3
> if (x>2) y <- 2*x else y <- 3*x
```

因为 $x > 2$ 为真， y 被赋值于 $2 * 3 = 6$ ，如果条件不为真时， y 被赋值于 $3 * x$ 。 if() 语句通常被用于自定义的函数中，下面是一个典型的例子。

例 4.4

在研究中经常会使用 cor() 函数来计算两个向量之间的相关系数，从中可以得到两者的线性关系。我们也可以写一个程序，同时给出相关系数和散点图。

```
> corplot <- function(x, y, plotit) {
+ if (plotit == TRUE) plot(x, y)
+ cor(x, y)
+ }
```

下面我们用一个例子将这个函数应用于两个向量。

```
> corplot(c(2,5,7), c(5,6,8), FALSE)
```

```
[1] 0.953821
```

例 4.5

下面的函数是基于埃拉托色尼筛（sieve of Eratosthenes）的求质数程序，这种方法是已知最古老的找出小于给定值 n 之内质数的方法。它的基本算法是：因为合数总是可以分解成若干个质数的乘积，那么如果把质数的倍数都去掉，剩下的就是质数了。因为 2 是质数，所以第一步把 2 的倍数去掉，此时 3 没有被去掉，可认为是质数，下一步再把 3 的倍数都去掉。再下一步时，因为 4 是 2 的倍数，在前一步时已经被清理掉，所以下一个数是 5 ，所以这一步会将所有 5 的倍数都消掉。下面的过程以此类推。

```
> Eratosthenes <- function(n) {
+ if (n >= 2) {
+   sieve <- seq(2, n)
+   primes <- c()
+     for (i in seq(2, n)) {
+       if (any(sieve == i)) {
+         primes <- c(primes, i)
+         sieve <- c(sieve[(sieve %% i) != 0], i)
+       }
+     }
+   return(primes)
```

```
    + } else {
        + stop("Input value of n should be at least 2.")
    + }
+ }
```

下面给出几个应用实例：

```
> Eratosthenes(50)
```

```
[1] 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

```
> Eratosthenes(-50)
```

```
Error in Eratosthenes(-50) : Input value of n should be at least 2.
```

这段程序的目的是找出小于给定数值 n 的所有质数。它的基本算法是下面这些命令：

```
sieve <- seq(2, n)
primes <- c()
for (i in seq(2, n)) {
    if (any(sieve == i)) {
        primes <- c(primes, i)
        sieve <- sieve[(sieve %% i) != 0]
    }
}
```

变量 *sieve* 包含了所有候选数，它是一个包含从 2 到 n 的整数向量。变量 *primes* 的初始值设为空，最后它将被用来存储小于 n 的所有质数。在变量 *sieve* 中的合数被消除后，剩余的质数被拷贝到 *primes* 变量中。

下面判断从 2 到 n 之间的每一个整数 i 是否依然包含在向量中。如果至少有一个逻辑变量元素为``真"，函数 any() 就返回一个真值。只要 i 依然保留在向量 *sieve* 中，那么它就一定是质数，因为它是还没有被消掉的最小值。凡是 i 的倍数都被消掉，因为它们是合数。假如 *sivev* 中所有的元素都是 i 的倍数，那么表达式 $(sieve\%\%i) == 0$ 就返回一个真值。但因为我们需要消除这些 i 的倍数，并保存其他数，所以用 ! 来对上面这个表达式取反，即 $!(sieve\%\%i == 0)$ 或者 $sieve\%\%i! = 0$ 。这样就可以用 $sieve < -sieve[(sieve\%\%i)! = 0]$ 消掉所有 i 的倍数。

上面这命令也会消掉 i 值本身，不过它已经被提前保存到 *primes* 变量中了。如果设定的 n 值小于 2，那么函数的输出就没有意义。为避免这种情况，当用户输入值小于 2 时，函数给出出错信息。

习题

1. 如果输入的 n 不是一个整数， *Eratosthenes*() 函数还能工作吗？此时会返回出错信息吗？
2. 用 *Eratosthenes*() 的编程思想证明质数有无穷多个。提示：假定所有的质数都小于 m，构建一个更大的值 n，它包含在 *sieve* 中，但没有被消掉。
3. 孪生质数 (twin prime) [1]是这样一对质数 (x, y)，它们满足 $y = x + 2$ 。请构建一个小于 1000 的孪生质数表。

[1]早在古代人们就知道质数有无穷多个，不过是不是存在无穷多个孪生质数，始终还是个猜想

4. 某银行提供一种投资组合，如果用户购买该投资产品小于等于三年的话，每年的利率是 4%（按年复利计算）。如果购买期限超过三年，则利率为 5%，假设某客户初始投资额为 P，投资年限为 n，则投资收益为 $I = P((1+i)^n - 1)$。请用程序计算该收益。

5. 抵押贷款利率有时候取决于抵押期开始还是结束。每月的还款可以用下面的公式来表示：

$$R = \frac{Pi}{1-(1+i)^{-n}}$$

其中 R 是每月的还款数，i 是每月利率（复利），P 是最初的本金，n 是按月表达的还款期限。用 *open* 作参数，如果 $open = TRUE$，那么 $i = 0.005$，此外 $i = 0.004$。

4.1.3 WHILE() 循环

有时候，程序员希望复执行某一个命令，但他并不提前知道应该重复的次数。这时候需要判断条件是否符合，如果符合就继续执行该命令。 while() 命令就可以用来实现这个功能。

这个命令的语法为： while(条件){ 命令}

该命令首先判断条件是否成立，如果条件不成立，就不执行任何命令，如果条件为``真"，就执行后面的命令，然后再一次判断条件是否还成立。这个过程会一直重复下去。

例 4.6

如果想要列出所有小于 300 的斐波那契数，但并不知道这个数列到底有多少个，所以我们不知道何时停止循环。此时可以用 while() 命令：

```
> Fib1 <- 1
> Fib2 <- 1
> Fibonacci <- c(Fib1, Fib2)
> while (Fib2 < 300) {
+ Fibonacci <- c(Fibonacci, Fib2)
+ oldFib2 <- Fib2
+ Fib2 <- Fib1 + Fib2
+ Fib1 <- oldFib2
+ }
```

这段命令的重点在下面这两条语句：

```
while (Fib2 < 300) {
Fibonacci <- c(Fibonacci, Fib2)
```

它的功能在于，只要创建并保存在 $Fib2$ 中的最后一个斐波那契数小于 300，那么就继续生成下一个数。不过这要求变量 $Fib2$ 中保存的是最新生成的斐波那契数，这可以通过下面的语句来刷新这个列表：

```
Fib2 <- Fib1 + Fib2
```

现在 $Fib2$ 是最新生成的斐波那契数，由于生成下一个数时，需要把先前的 $Fib2$ 转换为 $Fib1$，但前面的语句显然会把原先的 $Fib2$ 值冲掉，为防止这个现象发生，在执行上述语句之前先要把 $Fib2$ 复制备份为 $oldFib2$，然后再把 $oldFib2$ 赋值给 $Fib1$。

为了使上面的过程能执行，首先要对 $Fib1$， $Fib2$ 和 $Fibonacci$ 等变量赋初始值，在这个循环之中，这些对象都会被用到。

查看计算的结果：

```
> Fibonacci
```

```
[1] 1 1 1 2 3 5 8 13 21 34 55 89 144 233
```

请注意，随着 $Fibonacci < -c(Fibonacci, Fib2)$ 命令被循环执行，向量的长度会逐渐增加。如果这个数值很大的话，在 for() 和 while() 循环中就应该避免。因为 R 将不得不保留新产生的向量以及比最后一个元素更大的每一个元素，这样的结果将使程序执行效果大大降低。

习题

1. 在上面的例子中，变量 $oldFib2$ 并不是必须的。请只用 $Fib1$ 和 $Fib2$ 重写上面的 while() 循环程序，
2. 事实上， $Fib1$ 和 $Fib2$ 也不是必须的，请只使用 $Fibonacci$ 变量来构建这个循环。
3. 小于 $1,000,000$ 的斐波那契数共有多少个？
4. 依据 4.2 节车行卖车的例子中，用 while() 函数构建循环，计算不动点方程中的利率 i 。

   $i = \left(1 - (1+i)^{-20}\right)/19$

   i 的初始估计值使用 0.006 ，当连续的两次计算结果差异小于 0.000001 时，停止循环。假如换一个初始估计值情况会怎么样？
5. 在上面的习题中，请编写程序，同时能给出迭代的次数。

4.1.4 牛顿法解方程

牛顿方法被广泛应用于求解代数方程的根。已知方程为 $f(x) = 0$ ，假如 $f(x)$ 可导，其导数为 $f'(x)$ ，那么只要所选的初始估计值足够接近于真实解，通过下面的迭代过程就能收敛于方程的根。

设 $x_0 =$ 初始估计值

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$

这个算法的思想来自于泰勒级数近似：

$$f(x_n) \approx f(x_{n-1}) + (x_n - x_{n-1}) f'(x_{n-1}) \tag{4.1}$$

牛顿方法相当于设 $f(x_n) = 0$，然后使用（4.1）来求解 x_n。即使（4.1）只是一种近似方式，我们还是希望 x_n 能给出一个接近方程的根。要注意的是，在很多情况下即使 x_n 的初始估计值已经很接近方程的根，但还是没法收敛，甚至有些情况下不管初始估计是多少，都无法收敛。

例 4.7

假定方程 $f(x) = x^3 + 2x^2 - 7$ ，那么如果 x_0 足够靠近于方程三个根之一：

$$x_n = x_{n-1} - \frac{x_{n-1}^3 + 2x_{n-1}^2 - 7}{3x^2 + 4x}$$

就会收敛于方程的根。

使用下面的 R 命令可以求解上面的方程：

```
> x <- x0
> f <- x^3 + 2*x^2 -7
> tolerance <- 0.000001
> while (abs(f) > tolerance) {
+ f.prime <- 3*x^2+4*x
+ x <- x-f/f.prime
+ f <- x^3 + 2*x^2 -7
+ }
> x
```

上面的程序中，首先让 x 等于初始值 x_0，然后求 $f(x_0)$ 的值。只要 $|f(x_i)|$ 的值大于 0.000001，就用计算所得的值取代原 x，进行新一轮循环。请注意，程序中不需要添加变量 i，因为牛顿法是一个迭代算法，它能从前面的计算结果中算出新的结果来，所以没必要知道哪一次过程得到了现在这个结果。

习题

1. 方程 $x^7+10000x^6+1.06x^5+10600x^4+0.0605x^3+605x^2+0.0005x+5$ 只有一个实根。如果初始估计值是 $x=0$，那么用牛顿迭代法的话，需要多少步才能找到这个根？

2. 用牛顿法找到方程 $f(x)=x^4+3x^3-2x^2-7$ 等于零时的根。假定初始估计值是 $x=1$。

3. 寻找使方程 $f(x)=\cos(x)+e^x$ 等于零的一个根，使用初始估计值 $x=-1.5$。

4. 找到方程 $f(x)=(x-3)^4+7(x-2)^2+x$ 的最小值。

5. 请问有多少根能使方程 $f(x)=\dfrac{5x-3}{x-1}$ 等于零。请描述用牛顿法求解的步骤。假设初始估计值分别取 0.5， 0.75， 0.2 和 1.25。

6. 使方程 $f(x)=(x^2-6x+9)e^{-x}$ 为零的根有几个？分别是什么？描述牛顿法求解的步骤，并分别用初始估计值为 3， 3.2， 2.99 和 3.01 来求解。

7. 回到 4.2 节中车行卖车利率的问题，$i=(1-(1+i)^{-20})/19$，假设初始估计值取 $i=0.006$，用牛顿法求解，要迭代多少次才能使连续两次结果之间的差异小于 0.000001？

4.1.5 REPEAT 循环以及 BREAK 和 NEXT 命令

有时候程序中不需要运行固定次数的循环，也不希望使用 while() 函数进行判断。在这种情况下可以用 *repeat* 命令来创建循环，这个循环过程会一直进行，直到执行 *break* 命令来中断它。

该命令语法为：repeat{命令}

这个命令的特点是不断重复执行某段程序，所以必须在命令中包含一个中断条件，否则程序将陷入死循环：

if (条件) break

不过并不是必须包括中断条件，break 命令能使循环马上中断，这个命令同样也可以用于 for() 和 while() 循环中。next 命令能让程序马上跳转到循环头，重新执行循环。这个命令也可以用于任何一种循环语句中。

repeat 循环以及 break 和 next 命令并不常用。其实在循环中加入判断或者设定退出循环的条件，能让程序代码更容易理解。

例 4.8

下面用 repeat 循环来重写上面的牛顿迭代程序：

```
> x <- x0
> tolerance <- 0.000001
> repeat {
+   f <- x^3+2*x^2-7
+   if (abs(f) < tolerance) break
+   f.prime <-3*x^2+4*x
+   x <- x-f/f.prime
+ }
> x
```

这种方法可以少写一行计算 f 的语句。

习题

1. 另一种寻找方程解的算法是两分法。这个算法从两个初始估计值 x_1 和 x_2 开始，它们使得 $f(x_1)$ 和 $f(x_2)$ 具有相反的正负号。假设函数 $f(x)$ 是连续函数，那么在 x_1 和 x_2 之间必然存在方程的根。取 $f(x_1)$ 和 $f(x_2)$ 的中间点，并求 $f(x)$ 在这一点的值，选择仍然包含方程根的那一半区间，重复上面这个过程，直到得到满意的解。

   - 用两分法求方程 $f(x) = x^3 + 2x^2 - 7$ 的根。已知方程的根在区间 [0,2] 之间。要求精确度达到小数点后 6 位。
   - 假如方程在 0 和 2 分别取不同符号的值，并且在此区间上方程是连续的，那么证明此算法能保证收敛于方程的根，并给出循环的次数。

2. 下面的程序是用 while() 函数来实施埃拉托塞尼筛的算法。

```
> Eratosthenes <- function(n) {
+     if (n >= 2) {
+     sieve <- seq(2, n)
+         primes <- c()
+         while (length(sieve) > 0) {
+             p <- sieve[1]
+             primes <- c(primes, p)
+             sieve <- sieve[(sieve %% p) != 0]
+         }
+         return(primes)
+     } else {
+         stop("Input value of n should be at least 2.")
+     }
+ }
```

   - 阅读程序，理解程序的操作过程。
   - 证明当 $p \geq sqrt(n)$ 时，剩余的所有数字都是质数。
   - 修改上述程序，用 *break* 命令来优先获得上面的结果。

4.2 使用函数来管理复杂程序

现实中的程序比书本上的例子要长得多。大部分的用户不可能将这些过程同时记住。所以寻找一种能降低程序复杂性的方法就非常重要。这些年来，人们已经寻找到多种程序管理的策略。在本节中，我们简单介绍这些有效的策略。

4.2.1 什么是函数

函数是一个目标明确的自足单元。简单说来，函数接受输入，对输入进行计算（可能会输出中间计算值，绘制图表或者引用其他函数），输出计算结果。如果输入输出都很明确，程序员便能判断本函数是否有错，如果没有错误，那么就可以转入其他问题了。

例 4.9

假如每年向银行账户中存入 R 单位的货币，银行的利率是每年 i 单位。那么在 n 年之后账户中的余额是多少？

已知 n 年后的总余额可以用下面的方式来计算：

$R(1+i)^{n-1} + \cdots + R(1+i) + R = R\frac{(1+i)^n - 1}{i}$

根据这个算法来构建一个 R 函数：

```
> annuityAmt <- function(n,R,i) {
+ R*((1+i)^n-1)/i
+ }
```

假设每年存入 $400，连续存 10 年，银行的年利率是 5% 。那么余额是：

```
> annuityAmt(10, 400, 0.05)
```

```
[1] 5031.157
```

在各种计算机语言中， R 语言有一个特点，即它的函数是一个可操作的对象，就像向量和矩阵等普通对象一样。

一个函数的定义通常包含下面几个方面：

1. 函数名
2. 一对圆括号，其中包含了参数列表。当然参数也可以为空。
3. 一条命令，或者包含在一对花括号内的一系列命令。

像其他 R 对象一样，函数通常也需要命名。用户可以为自己定义的函数选择一个合适的名字，以便能很好地描述函数的功能。例如， var() 函数用于计算方差， median() 函数用于计算中位数。函数的命名很重要，假如连自己都记不住，那以后要调用它时可能不得不花很长的时间来搜索它。假如选择的函数名带有歧义，那么这段程序在以后会很难理解。例如，假设在上面例子中定义的函数名为 annuityRate 而不是 annuityAmt ，那么其他读者将误解这个函数的功能。

当 R 执行函数定义时，它能产生一个包含三部分的对象，即函数头，函数体和函数所执行的环境参量。

函数定义中的前面两项用于创建函数头。一个 R 函数的头部描述了输入或者 ``参数"。例如前面定义的函数 Eratosthenes ，其头部是 function(n) 。它告诉用户函数带有一个参数 n ，它规定了这个筛的上限。函数 annuityAmt 的头部是 function(n,R,i) ，它表示这个函数带有三个参数，分别是 n ， R 和 i 。函数可以包含任意数目的参数。

为了便于用户使用函数，可能会选用一些默认参数，这样当用户没有给出指定值时，函数就选择默认值。例如，可以用下面这样的函数头：

annuityAmt <- function(n, R = 1, i = 0.01)

它表示，如果用户调用函数 annuityAmt(24) ，但没有给出 R 和 i 的值时，函数就选择 $R = 1$ 和 $i = 0.01$ 为默认值来计算。

函数的第二部分是函数体。它是一条命令或者包含在花括号之内的多条命令。它规定了函数要执行的计算。在 Eratosthenes 函数的例子中，函数体是：

```
{
    if (n >= 2) {
        sieve <- seq(2, n)
        primes <- c()
        for (i in seq(2, n)) {
            if (any(sieve == i)) {
                primes <- c(primes, i)
                sieve <- c(sieve[(sieve %% i) != 0], i)
            }
        }
        return(primes)
    } else {
```

```
        stop("Input value of n should be at least 2.")
    }
}
```

在函数体中，通常会在某些位置放置一个诸如 return(primes) 这样的命令，它会返回函数的输出值。在 R 之中，所有的函数都只有一个输出值，而在某些计算机语言中，可以没有输出，也可以有多个输出。如果在函数体中没有放置 return() 命令，那么函数就返回它的最后计算值。比如 annuityAmt 函数就是这样做的。

函数的第三部分 -函数的环境 -是最难理解的，因为它是最具体的。我们在这里不打算给详细的描述，仅限于下面这样的定义：环境是对在其内所定义函数的参考。

在此做一些说明，在 Eratosthenes 函数中，用了两种很不相同的对象： n ， *sieve* ， *primes* 和 i 都是在函数内部被定义的局部变量，对它们的含义没有歧义。但是像 *seq* ， *c* ， *any* ， *return* 以及 *stop* 它们并没有在函数内被定义，它们是 Eratosthenes 所处的 R 环境中的一部分。通常来说它们都是函数（当然并不总是这样），但局部变量则不是。

真正有意思的是，在 Eratosthenes 函数中，我们可能已经定义了一个新函数。它的环境可以包括 n ， *sieve* ， *primes* 和 i 。例如我们希望用一个函数 noMultiples 消去质数的倍数。

```
> Eratosthenes <- function(n) {
+ if (n >= 2) {
+   noMultiples <- function(j) sieve[(sieve %% j) != 0]
+       sieve <- seq(2, n)
+       primes <- c()
+       for (i in seq(2, n)) {
+           if (any(sieve == i)) {
+               primes <- c(primes, i)
+               sieve <-c(noMultiples(i), i)
+           }
+       }
+       return(primes)
+   } else {
+       stop("Input value of n should be at least 2.")
+   }
+ }
```

noMultiples 函数在其函数头部定义了 j ，所以 j 就是一个局部变量， *sieve* 则在它的环境之中。

习题

1. 证明诸如 *var* , *cos* , *median* , *read.table* 和 *dump* 等对象都是函数。
2. 假设某人在银行存入数量为 P 的钱，复利是 $i.r$ 。如果他希望在银行存 n 个计息周期，那么总的余额是 $P(1+i.r)^n$ 。
   - 写一个 R 函数 compund.interest() 来计算这个余额。这个函数应该包含函数的三个部分。
   - 通常一个计息周期是一个月，在这个例子中，如果储户开始时存入 $1000 ，每月利息是 1% ，那么在存入 30 个月后的总额会是多少？
3. 使用两分法来计算由用户自己提供的一个方程的根。

4.2.2 变量范围

变量的范围是指变量在什么地方能被识别。例如，在函数内部定义的变量，它们只有在函数内部才能被识别。在不同的函数中，用户可以创建名字相同的变量，它们之间并不会相互冲突。

例 4.10

在这个例子中，创建了两个函数 f 和 g，它们两者都包含局部变量 x。f 会调用函数 g，并改动 g 中的变量 x，但这并不会影响 f 中 x 的值。

```
> f <- function() {
+ x <- 1
+ g() # 函数 g 不会影响局部变量 x
+ return(x)
+ }
> g <- function() {
+ x <- 2 # 修改函数 g 中的 x 并不是 f 中的 x
+ }
> f()
```

```
[1] 1
```

在 R 之中，变量的范围由函数的环境来控制。在控制台所定义的是全局变量，它能被所有用户自定义的函数所调用。在函数内部定义的变量，只是在函数内部可见，也只能在函数内部发挥其功能。

使用局部变量对于优化程序非常重要，因为程序员不必担心变量值在函数的其他地方被修改。程序员很容易给出变量的赋值，并且一旦通过编译后，就能信任其输出结果。

R 也包含一些程序包，这些程序包中包含了函数和数据。通常这些程序包有他们自己的使用范围（称为命名空间）。不过对命名空间（name spaces）的详细说明，以及那些没有命名空间的程序包的说明，都超出了本书的范围。

4.3 编程技巧

4.3.1 使用 $FIX()$ 函数

在编写函数时，程序员常常犯一些错误。函数 fix() 能帮助用户用一个新的函数替换不正确的版本，它也能用于创建一个新函数。

例 4.12

输入 fix(Eratosthenes) 来修改 Eratosthenes 函数.

例 4.13

输入 fix(factorial) 创建一个新的函数，名叫 factorial （阶乘）。现在可以输入函数的命令语句和函数体。首先要创建一个对象用于计算阶乘，我们可以使用 prod() 函数。这个函数将其所有的元素相乘，所以 prod(1:n) 应该等于 $n!$。然后把这个命令包含在程序体中，包含在一对花括号之内。其次创建程序头，它应该是 function(n) 。退出编辑器之后，可以测试一下这个函数。

4.3.2 用 # 写程序注释

在编写函数时，用户可以用 # 符号插入注释，比如函数的名称和功能描述等。在 R 执行程序时，它会忽略从 # 开始直到行尾的所有内容。

对自己创建的新函数添加名称和注释是一个很好的编程习惯，这样在以后能很方便地记忆起编写程序时的初衷，也便于其他读者理解程序内容。因为人们很容易忘记数星期前编写函数时的基本想法。

有时候对某一行程序添加注释也非常有用，特别是使用了一些很模糊的命令时更加有用。

习题

1. 计算 $10!, 50!, 100!$, 和 $1000!$ 。

2. (a) 用 fix() 和 factorial() 创建一个函数，用于计算二项式系数 $\binom{n}{m}$

   (b) 计算 $\binom{4}{2}$ ， $\binom{50}{20}$ 和 $\binom{5000}{2000}$ 。

   (c) 求和函数 sum() 用于对所有元素相加求和，自然对数函数 log() 和指数函数 exp() 分别取变量的自然对数和自然指数。请用这两个函数改善上述求二项式系数的函数。

   (d) 用改善的函数计算 $\binom{4}{2}$ ， $\binom{50}{20}$ 和 $\binom{5000}{2000}$ 。

3. 参照 4.2.1 节习题 2 中的 compound.interest() 函数。通常利率表达为每年的名义利率，按每月记复利。要得到每月的名义利率，只要将年利率除以 12 。更一般的方法是，假如一年有 m 个计息周期，那么每个周期的名义利率等于年利率除以 m 。请修改 compound.interest() 函数，用 j 表示年利率，用 m 表示计息周期，则函数体中每个计息周期的有效利率 $i.r$ 就等于 j/m 。（您需要从参数中删除 $i.r$ ）。

4. 使用新编辑的 compound.interest() 函数，计算年利率为 12% ，初期存款为 1000 ，每天记一次利率的话，一年后的总额是多少。如果计息周期分别为月和年的话，结果又是如何？请比较结算结果。

5. 假设吴女士希望加入一个住房抵押贷款项目。她想知道每个时期需要付多少钱。如果 P 是首付金额，$i.r$ 是实际利率， n 是的贷款时间长度。那么每个周期的付款额 R 是：$R = \dfrac{Pi.r}{1-(1+i.r)^{-n}}$.

   (a) 构建一个函数 mortgage.payment() 来表达这个支付公式。

   (b) 假设吴女士的首付金额是 100000 ，利率是 1% ，计息周期是 300 。那么她每月应该支付多少？

   (c) 使用 annuity() 函数计算 10 年后的累加总额。假设每个周期付款 400 圆，在这个期间，利率从 0.01 按每次增加 0.01 的幅度增加到 0.20 。

4.4 一些通用编程指南

编写计算机程序通常可以被分解为下面这些步骤：

1. 理解问题。

2. 解决问题的基本思路。

3. 将基本思路细化为具体的执行步骤。

4. 检查程序是否能执行，并得到预期结果。如果结果满意则完成此项目，如果不够完善，则返回到第二步。

例 4.13

下面我们将编写一个程序，将一个向量按升序排列。

1. 理解问题。要理解目前所面对的问题，一个好的办法是举一个具体的例子。可以考虑一些简单的例子，当然也不能太简单。在这个问题中，假定要排序的向量为 $3, 5, 24, 6, 2, 4, 13, 1$ 。那么用户需要写一个函数，名为 sort() ，它能实现下面的功能：

```
x <- c(3, 5, 24, ..., 1)
sort(x)
```

   程序的输出应该是按升序排列的数列： $1, 2, 3, 4, 5, 6, 13, 24$ 。

2. 解决问题的基本思路。第一个想法是寻找哪一个是其中最小的数字，并把它取出放在一边，然后在剩余的数字中不断重复这个过程。

   另一个想法是，从头至尾比较相邻的两个数字，如果他们顺序不对，就将他们位置互换。不过经过检验后发现，第二个方法是无效的。假设需要排序的数列是 $2,1,4,3,0$，比较相邻数据大小之后，得到的排序结果为 $1,2,3,0,4$。再检验一下这种算法，结果发现最大的数总能排到最后，这意味着在进行一次排序后，可以对除最后一位数字外的剩余数字再次排序，直到遍历所有的元素。

3. 详细的实施。在实施阶段，需要解决特定编码的问题。在这个排序问题中，需要解决的问题是：如何将 $x[i]$ 和 $x[i+1]$ 的位置互换。下面给出一个例子将 $x[3]$ 和 $x[4]$ 的位置互换。

```
> save <- x[3]
> x[3] <- x[4]
> x[4] <- save
```

   请注意，在将 $x[3]$ 寄存到别处之前，不可以用 $x[4]$ 来替换 $x[3]$，因为 $x[3]$ 的值被冲掉后，就没法再把它赋值给 $x[4]$ 了。现在已经准备好来写这个排序函数了：

```
> sort <- function(x) {
+ for(last in length(x):2){
+     for(first in 1:(last-1)) {
+         if(x[first] > x[first + 1]) {
+             save <- x[first]
+             x[first] <- x[first + 1]
+             x[first + 1] <- save
+         }
+     }
+ }
+   return (x)
+ }
```

4. 检验。用一个简单的例子来检验所编写的程序是否有错。

```
> sort(c(2, 1))
```

```
[1] 1 2
```

```
> sort(c(2, 24, 3, 4, 5, 13, 6, 1))
```

```
[1] 1 2 3 4 5 6 13 24
```

   读者还可以试一些其他例子，比如当向量的长度为 1 的时候，结果会如何呢？

```
> sort(1)
```

```
Error in if (x[first] > x[first + 1]) {: missing value where
TRUE/FALSE needed
```

产生这个问题的原因在于，当 length(x)==1 时， *last* 变量的值会变成 1 : 2 ，而事实上应该不赋值。不过这个问题不需要对程序重新编写，只要在程序开头对这一特殊例子作一个说明即可。

```
> sort <- function(x) {
+ if (length(x) < 2) return (x)
+ for(last in length(x):2) {
+     for(first in 1:(last - 1)) {
+         if(x[first] > x[first + 1]) {
+             save <- x[first]
+             x[first] <- x[first + 1]
+             x[first + 1] <- save
+         }
+     }
+ }
+     return (x)
+ }
```

重新测试一下这个新函数：

```
> sort(1)
```

```
[1] 1
```

结果很成功，或者至少达到预期目的。您还可以试试，看看我们还错过了什么。

4.4.1 自上而下设计

设计出一个详细执行的程序看起来是一项艰巨的任务。这项任务能顺利执行的关键在于，要将一个大的任务分解成若干个能解决的小任务。一个有效的方法是 ``自上而下的设计"，这种设计方法类似于在写文章之前先列出提纲，然后再填充详细的内容：

1. 把整个程序划分成几个主要步骤。
2. 把每个主要步骤细化为若干小阶段。
3. 重复上述过程，直到完成一个程序。

例 4.14

在例 4.13 中使用的排序算法被称为 ``冒泡法"。这种算法容易编程，而且当向量 $x$ 不是很长时还挺有效。但是当 $x$ 很长时候，就需要更高效的算法了，其中有一种算法称为 ``合并排序"。

合并排序的基本原理是，将一个需要排序的向量拆分成两部分，每部分分别排序，然后将两者合并到一起。在合并过程中，只需要比较每部分第一个元素的大小，然后选出所有元素中最小的那个，并将这最小的元素移除。此时在剩余的部分中，第二个元素就成了最小。我们可以将两部分合并成一个有序的向量。

那么怎么对每一部分进行排序呢？一种选择是继续使用冒泡法，当然更好的方法是对这每一部分依然用合并排序，这是一种递归算法。我们编写的 mergesort() 函数可以调用自身。因为变量作用域的原因，函数每调用一次，局部变量就被复制一次，它们之间不会相互干扰。

算法理解：

在进行程序设计之前，可以使用一些简单的数值例子来帮助理解算法。例如，假定 x 是向量 [8,6,7,4] ，我们希望排序后的结果存储在向量 R 之中。那么合并排序过程就应该是这样的：

1. 将 x 拆分成两部分： $y \leftarrow [8,6]$ ， $z \leftarrow [7,4]$ 。

2. 对 y 和 z 分别排序： $y \leftarrow [6,8]$， $z \leftarrow [4,7]$。

3. 将 y 和 z 合并：

   (a) 比较 $y_1 = 6$ 与 $z_1 = 4$： $r_1 \leftarrow 4$，移除 z_1，现在 z 变为 [7]。

   (b) 继续比较 $y_1 = 6$ 与 $z_1 = 7$： $r_2 \leftarrow 6$，移除 y_1，现在 y 变为 [8]。

   (c) 比较 $y_1 = 8$ 与 $z_1 = 7$： $r_3 \leftarrow 7$，移除 z_1，现在 z 为空。

   (d) 把剩余的 y 值存放到 R 中： $r_4 \leftarrow 8$。

4. 输出结果 $r = [4,6,7,8]$。

把想法翻译成代码

一个有效的方法是，把编写代码的过程看成一步一步地优化程序直到它生效的过程。

可以从一句总体的描述开始，每一部分逐步扩大。使用两个注释符号 ## 来标注那些仍然需要扩展的的部分。给这些注释编号，以便于在以后引用它们，不过在实际编程中，您可能不需要这么做。在把这些部分扩充后，将注释符号改回到普通的格式。

这个程序的目的只有一个，我们可以将它作为第一条注释。

```
## 1. 使用合并排序法对向量排序
```

依据上面的步骤逐步扩展，将详细的内容添加进去。第一步是这样简单的扩展，即首先需要输入一个向量 x，这样才可以用定义的函数 *mergesort* 来处理它。随后对这个向量排序，最后希望函数返回并输出计算结果。

```
# 1. 使用合并排序法对向量排序
mergesort <- function (x) {
## 2: 对 x 排序，并输出结果
}
```

接下来扩展第二步，请注意这个合并和排序的过程是怎么实现的：

```
# 1. 使用合并排序法对向量排序
mergesort <- function (x) {
# 2: 对 x 排序，并输出结果
## 2.1: 把 x 分拆成两部分
## 2.2: 分别对每一部分排序
## 2.3: 排序后的两部分合并成一个有序向量
return (result)
}
```

上面的每一个子步骤都需要扩展，首先来看第 2.1 步:

```
# 2.1: 把 x 分拆成两部分
len <- length(x)
x1 <- x[1:(len / 2)]
x2 <- x[(len / 2 + 1):len]
```

请特别注意那些``边界" 问题，一般来说，希望排序的向量会有多于一个元素，但是作为一个排序函数它也应该能处理只拥有一个元素向量的排序。上面的程序并不能处理 $len < 2$ 的情况。

所以必须再次修改第 2.1 步。解决方法很简单，如果 x 的长度是 0 或者 1，那么函数就直接返回 x 向量。如果不是的话，还依照上面的方法来拆分。不过这已经超出了在 2.1 中的任务了，所以需要对任务重新描述，下面是重新定义的任务步骤：

```
# 1. 使用合并排序法对向量排序
mergesort <- function (x) {
    # 检验向量长度，看它是否需要排序
```

```
    len <- length(x)
    if (len < 2) result <- x
    else {
      # 2 对 x 排序，并输出结果
      # 2.1 把 x 分拆成两部分
      y <- x[1:(len / 2)]
      z <- x[(len / 2 + 1):len]
      ## 2.2 分别对每一部分排序
      ## 2.3 排序后的两部分合并成一个有序向量
    }
    return(result)
}
```

第 2.2 的扩展非常简单，因为它可以直接调用 mergesort() 函数，即使我们现在还没有写出这个函数。假定完成这个函数设计后，那么显然就能使用这个设计成果。所以第 2.2 步就可以写成：

```
# 2.2: 对 y 和 z 排序
y <- mergesort(y)
z <- mergesort(z)
```

第 2.3 步将会比较复杂，所以要加倍仔细。因为函数最后会返回一个向量 result ，那么剩余的步骤该包括哪些呢？现在将整个函数都重新写一遍：

```
# 1. 使用合并排序法对向量排序
mergesort <- function (x) {
    # 检验向量长度，看它是否需要排序
    len <- length(x)
    if (len < 2) result <- x
    else {
      # 2 对 x 排序，并输出结果
      # 2.1 把 x 分拆成两部分
      y <- mergesort(y)
      z <- mergesort(z)
      ## 2.2 对每一部分排序
      y <- x[1:(len / 2)]
      z <- x[(len / 2 + 1):len]
      # 2.3 排序后的两部分合并成一个有序向量
      result <- c()
      ## 2.3.1 while 每一堆中都有剩余数据()
      ## 2.3.2 将第一个最小值放到最后位置
      ## 2.3.3 把上面的值从 y 和 z 中删除
      ## 2.3.4 把剩余部分放到结果的最后部分
    }
    return(result)
}
```

现在，最后几步也比较容易扩展了。第 2.3.2 和 2.3.3 要交织进行，因为它们两者都依赖于比较 $y[1]$ 和 $z[1]$ 中哪个一最小。

```
# 1. 使用合并排序法对向量排序
mergesort <- function (x) {
    # 检验向量长度，看它是否需要排序
    len <- length(x)
    if (len < 2) result <- x
    else {
```

```
    # 2 对 x 排序，并输出结果
    # 2.1 把 x 分拆成两部分
    y <- mergesort(y)
    z <- mergesort(z)
    ## 2.2 对每一部分排序
    y <- x[1:(len / 2)]
    z <- x[(len / 2 + 1):len]
    # 2.3 排序后的两部分合并成一个有序向量
    result <- c()
    ## 2.3.1 while 每一堆中都有剩余数据()
    while (min(length(y), length(z)) > 0) {
    ## 2.3.2 将第一个最小值放到最后位置
    ## 2.3.3 把上面的值从 y 和 z 中删除
    if (y[1] < z[1]) {
      result <- c(result, y[1])
      y <- y[-1]
    } else {
      result <- c(result, z[1])
      z <- z[-1]
    }
  }
    ## 2.3.4 把剩余部分放到结果的最后部分
  if (length(y) > 0)
    result <- c(result, y)
  else
    result <- c(result, z)
  }
  return(result)
}
```

习题

1. 修改本节中所描述的 mergesort() 函数，使之包含一个逻辑变量，当这个值为``真" 时，函数进行降序排序。

2. 已知函数

   $$\begin{aligned} f(x,y) &= 0 \\ g(x,y) &= 0 \end{aligned}$$

   使用牛顿法能得出数值解。设定初始估计值 x_0 和 y_0。然后执行下面这样的迭代，其中 $n = 1, 2, \cdots$；

   $x_n = x_{n-1} - (g_{y,n-1}f_{n-1} - f_{y,n-1}g_{n-1})/d_{n-1}$

   $y_n = y_{n-1} - (f_{x,n-1}g_{n-1} - g_{x,n-1}f_{n-1})/d_{n-1}$

   其中：

   $f_{x,n-1} = \frac{\partial f}{\partial x}(x_{n-1}, y_{n-1})$

   $f_{y,n-1} = \frac{\partial f}{\partial y}(x_{n-1}, y_{n-1})$

   $g_{x,n-1} = \frac{\partial g}{\partial x}(x_{n-1}, y_{n-1})$

   $g_{y,n-1} = \frac{\partial g}{\partial y}(x_{n-1}, y_{n-1})$

   $f_{n-1} = f(x_{n-1}, y_{n-1})$

   $g_{n-1} = g(x_{n-1}, y_{n-1})$

   $d_{n-1} = f_{x,n-1}g_{y,n-1} - f_{y,n-1}g_{x,n-1}$

   当函数值趋近于 0 时，迭代结束。

(a) 写出进行迭代的函数

(b) 将这个函数用于计算下面的方程组：

$$\begin{aligned} x + y &= 0 \\ x^2 + 2y^2 - 2 &= 0 \end{aligned}$$

计算这个方程组的解，用于检验您的数值解。

4.5 调试与维护

程序中的错误被称为``臭虫"(bug)，所以修改错误的过程就被称为捉臭虫的过程 (debug)，我们把这个过程称作程序调试。显然调试过程是很困难的，所以最好是程序中不包含``臭虫"，不过犯错是不可避免的。

下面这些步骤能有效地协助发现和修正程序中的错误：

1. 发现错误的存在。
2. 使错误重现。
3. 查明错误的原因。
4. 修正错误和测试。
5. 查询类似的错误。

下面详细讨论一下这五个步骤。

4.5.1 发现错误的存在

有时候发现程序中的错误很简单，比如程序不执行，显然就是有错误。然而，另外一些时候，程序似乎执行了，但是输出结果却不正确；或者对于某些输入来说，程序执行，但对另外一些输入，程序就不执行。这种类型的错误就比较难发现了。

有几种方法能简化这个过程。首先，依据前一节的建议，将程序分解成若干简单明确的功能块。描述它们的输入与输出。在函数内部可以测试输入与输出，看看是否符合假设和期望。

在某些情况下，可能值得写两个版本的函数：其中一个版本的执行速度可能慢一些，但能肯定是正确的，另一个版本可能执行速度较快，但不怎么确定其正确性。作者可以在各种情况下测试这两种版本，使它们有相同的输出。

经验表明，对于只是在某些特定输入时出错的情况，往往是所谓``边缘案件"；中间的情况往往是正确的。所以要测试这些边缘案例。例如，测试向量长度为 0 的情况，测试特别大或者特别小的值，等等。

4.5.2 使错误重现

在修正一个错误之前，首先需要知道哪里发生了错误。如果能知道什么原因导致错误的发生，那么事情就会简单得多。但很多情况下这些错误是不可预知的，所以处理起来会相当困难。好在对于大多数电脑来说，都是可预测的，即相同的输入总会得到相同的输出。所以困难就在于找到哪些是必要的输入。

例如，在编程中一个常见的错误是写错了变量名。通常情况下，马上就能得到一个错误警告，不过有时候因为恰好选了一个现存的变量名，那么可能就会得出一些错误的结果。这些结果看起来像是随机的，因为它依赖于一些不相关的变量。要追踪这一类问题，关键在于仔细的工作。为了将事情尽量简化，打开一个新的 R 进程，看看您是否能重复这个错误。一旦可以重现这个错误，那么就能追踪到它。

有些程序是做随机仿真，在这种情况下，可以通过设定随机数种子来重现相同的仿真结果。具体请参阅第五章内容。

4.5.3 查明错误的原因

当确认了错误存在后，下一步就是查找它的原因。如果程序终止并给出了出错信息，那么就应该仔细阅读它们，试着理解这些错误信息的内容。

在 R 中，用户可以使用 traceback() 函数获取一些额外的错误信息。当错误发生后，R 将当前活动函数的堆栈信息保存起来，用户可以使用 traceback() 函数来打印这个列单。

例 4.15

下列函数将计算变量除以其均值后的标准差。然而对于 0 值会返回出错信息。

```
> cv <- function(x) {
+     sd(x/mean(x))
+ }
> cv(0)
```

```
Error in var(x, na.rm = na.rm) : missing observations in cov/cor
```

在返回的出错信息中，出现了 var()，可是前面的程序中并没有使用这个函数。为了弄清楚在何处调用了这个函数，使用 traceback() 来追踪：

```
> traceback()
```

```
3: var(x, na.rm = na.rm)
2: sd(x/mean(x))
1: cv(0)
```

信息显示，标准差的函数 sd() 调用了 var()，正是在这里出现了错误。当计算 $x/mean(x)$ 时，返回了错误的值。下面可以直接来测试一下这个过程。

```
> x <- 0
> mean(x)
```

```
[1] 0
```

```
> x/mean(x)
```

```
[1] NaN
```

解决的方法是，需要重新检查一下函数 cv() 的输入接口。方差系数的计算要求 x 的值为正，所以在计算之前必须先进行判断：

```
> cv <- function(x) {
+ stopifnot(all(x>0))
+ sd(x/mean(x))
+ }
> cv(0)
```

Error: all(x > 0) is not TRUE

当 $x > 0$ 不为真，需要调用内建标准函数 stopifnot() 来停止执行程序。

traceback() 函数能显示何处出现了错误，但它不能指出为什么会发生错误。而且，很多情况下，错误的程序也能运行，而不会返回出错信息，只不过让人觉得运算结果不太合理。那么此时应该怎么办呢？

通过适当的事先规划，可以在这一步上做得更容易一些。第 4.5.1 节中的一些建议可以在这里使用。如果已经选择有意义的变量名，那么可以将变量的值打印出来，看看他们是否包含了它应该包含的内容。要做到这一点，最简单的方法是在函数中添加这样的语句：

```
cat("In␣cv,␣x=", x, "\n")
```

这条命令将输出 x 的值，指出信息从哪里来。其中句末的（\n）是在打印后另起一行。或许有人更喜欢使用 print() 函数，因为它有更好的显示格式。不过它每次只能打印一个对象，所以用户可以这样来调用这个函数：

```
cat("In␣cv,␣x=\n")
print(x)
```

一个更灵活的方法是使用 browser() 或者 debug() 函数来查看函数值。请参阅第 4.5.6. 节。

另一个有效的方法是手工模拟函数进程，它能帮助理解类似小函数中的错误。可以把自己设想成 R 程序，写下每一步过程中变量的值。结合上面所说的若干调试技术，就能更好地理解 R 中的信息。假如手工模拟过程的结果和 R 的结果不同，那么可以确认 R 的过程和期望的过程不同。当然很多情况下是因为对 R 的过程不够了解，当然也可能确实发现了 R 中的一个缺陷。

4.5.4 修正错误和测试

一旦发现了程序中的错误，就需要修正它，而且不能产生新问题。随后要对程序重新测试，在测试过程中，需要包括那些容易导致错误的情况，包括边界条件等能考虑到的各种情况。

4.5.5 查询类似的错误

通常来说，一旦发现并修正了一个程序漏洞，就能发现其他类似的漏洞。所以应该将整个程序重新检查一遍，看看有没有类似的重复错误。因为有些错误犯了一次也有可能犯第二次。

4.5.6 BROWSER() 与 DEBUG() 函数

除了可以使用 cat() 和 print() 函数来调试程序外， R 还允许用户调用 browser() 函数。这个函数允许用户中断函数进行检查（甚至修改）局部变量，它也允许用户在评估函数环境时执行其他 R 命令。

用户可以执行下面这些调试命令：

- n - ``下一步"；执行程序中的下一条代码，它是逐条执行的。
- c - ``连续"；让程序连续执行。
- Q - 退出调试。

另一个进入调试环境的方法是调用 debug() 函数。例如用户要调试函数 f 就使用命令 debug(f) ，退出调试时使用 undebug(f) 命令。

例 4.16

现在回过头来看看前面曾经谈过的 mergesort() 函数。通过 debug() 函数，在所有的计算完成之前来查看 x 的值，然后在排序之前再次检查 y 和 z 的值。

```
> debug(mergesort)  # 设定程序调试标记
> mergesort(c(3, 5, 4, 2)) # 函数中断并显示返回值
debugging in: mergesort(c(3, 5, 4, 2))
debug: {
len <- length(x)
if (len < 2)
        result <- x
else {
        y <- x[1:(len/2)]
        z <- x[(len/2+1):len]
        y <- mergesort(y)
        z <- mergesort(z)
        result <- c()
        while (min(length(y), length(z)) > 0) {
                if (y[1] < z[1]) {
                        result <- c(result, y[1])
                        y <- y[-1]
                }
                else {
                        result <- c(result, z[1])
                        z <- z[-1]
                }
        }
        if (length(y)>0)
                result <- c(result, y)
        else result <- c(result, z)
}
return(result)
}
Browse[1]> x # 显示 x 的入口值
[1] 3 5 4 2
Browse[1]> n # 进入程序体
debug: len <- length(x)
Browse[1]> # 输入等价于给 len 赋值
debug: if (len < 2) result <- x else {
y <- x[1:(len/2)]
z <- x[(len/2+1):len]
... # 忽略显示 if 语句
Browse[1]>  n
debug: y <- x[1:(len/2)]
Browse[1]>  n
debug: z <- x[(len/2+1):len]
Browse[1]> n
debug: y <- mergesort(y)
Browse[1]> y # 调用 mergesort 函数之前查看 y 的值
[1] 3 5
Browse[1]> z # 查看 z
[1] 4 2
Browse[1]> Q # 退出
>  undebug(mergesort)  # 从 mergesort 中移除调试标记
```

4.6 有效编程方法

当编写程序代码后，可能会发现它比预期的运行速度要慢很多，即使现在计算机速度大大提高之后，还有这样的情况发生。不过总有提高程序运算速度的方法，这个方法就是优化。在本节中列举几个在 R 中手动优化程序的例子，即重写部分代码以提高运行速度。另一个方法是所谓自动优化，R 并不提供自动优化，不过在其他编程环境中有这种功能，当然还可以采用硬件优化的方法，即采用更高性能的计算机来运行这些程序。

优化需要付出一定的成本，手动优化需要耗费大量的时间，因为程序代码晦涩难懂，常常会遗漏一些错误。在优化程序之前，一定要确保程序的正确性，否则后期的调试将异常艰难。同时还需要对程序有一个正确的评价，即它是否值得进一步优化。

4.6.1 理解优化工具

为了编写高效的代码，首先需要了解程序的操作平台。例如 R 是按照向量操作来设计的。采用向量运算比对单个元素操作要快很多，例如计算两个向量的和可以用下面的方法：

```
> X <- rnorm(100000)    # Xi ~ N(0, 1) i=1, ..., 100 000
> Y <- rnorm(100000)    # Yi ~ N(0, 1) i=1, ..., 100 000
> Z <- c()
> for (i in 1:100000) {
+ Z <- c(Z, X[i] + Y[i]) # 这一步计算耗时 53 秒
+ }
```

显然这种算法在 R 中效率非常低。它要读取并分配 Z 向量 10 万次，每一次都使 Z 的长度增加一单位。既然我们早就知道 Z 的长度了，为什么不在一开始就分配好，以后只要修改其内容就行了。

```
> Z <- rep(NA, 100000)
> for (i in 1:100000) {
+ Z[i] <- X[i] + Y[i]  # 这一步计算耗时 0.88 秒
+ }
```

只是简单地避免多次分配向量长度，就把运算效率提高了大约 60 倍。

在 R 中，一个更自然的方法是采用向量来计算，例如：

```
> Z <- X + Y # 大约 0.002 秒
```

完全采用向量运算后，可提高效率达 440 倍，这样第一种方法提高了 26500 倍效率。做个比喻，如果刚开始的程序运算需要一年，现在就只要二十分钟了，这相当于实现了不可能实现的任务。

在上面的例子中，刚开始程序运行花费了 53 秒，优化后的程序运行花费了 2 毫秒，这是一个很大的比例，不过也只是节省了 53 秒而已。在这个例子中，修改之后的代码更清晰更明显，所以应该认为这种努力是值得的。但如果被迫花了很多时间在优化上，最后得到一些不容易阅读的代码，那么这样的努力就是不值得的。

4.6.2 使用高效率的算法

例 4.17

假设有一张表格来记录全班考试的成绩，总共有 n 张试卷，试卷上记录着姓名和成绩。还有一张全班名册表，如何将成绩记录在这个名单后面？

一个效率不高的算法：

1. 从名册表上读入第一个学生的姓名。
2. 从 n 张试卷中随机选出一张。
3. 如果试卷上的名字和名册上第一个学生的姓名相符，则登录成绩，然后读取第二个学生姓名。

4. 如果姓名不相符，则将试卷放回，再次随机抽取一份试卷。

5. 连续执行上述过程，直到所有成绩都登记完毕。

这个算法的效果如何？答案可以通过找到第一个相符的姓名所需的时间来衡量。

因为总共有 n 张试卷，所以找到正确试卷的概率是 $1/n$ 。找到第一个正确试卷前需要抽取试卷的次数是一个参数为 $1/n$ 的几何分布随机变量。这个随机变量的期望值是 n 。对于第二个学生来说，期望值将是 $n-1$ ，因为已经找到了一张正确的试卷。总的来说，完成整个搜寻所需的时间是 $\sum_{i=1}^{n} i = n(n-1)/2$ 乘以每次搜寻所需的时间。

一个改善后的算法：

1. 抽出第一张试卷。

2. 扫描名单，直到找到和试卷上一致的姓名，登录成绩。

3. 重复上述过程。

对每一张试卷来说，这种算法平均需要扫描一半的名单长度，所以消耗的总时间是 $n^2/2$ 乘以扫描第一个姓名所需的时间，加上 n 乘以抽取一张试卷所需的时间。即使这样，我们也需要 $n^2/2 + n = n(n+2)/2$ 个步骤，每一步都比上面的算法要快一点。

一个更快的算法：

1. 按字母顺序将名单册进行排序。

2. 抽取一张试卷。

3. 使用字母索引寻找姓名，登录成绩。

4. 重复上述过程。

这是一种更快的算法，因为使用二分法之后，从一张有序的列表中进行搜索要快得多：先将名单一分为二，记录中间位置的姓名字母，看试卷上的姓名是落在前半张名单还是落在后半张名单中。然后将这半张姓名表再一分为二进行搜索，持续上述过程直到找到相符合的姓名。找到每一张试卷的时间为 $\log_2 n$ ，所以总共花费的时间是 $n \log_2 n$ 。对于一个中等或较大的班级来说，这种算法比 $n^2/2$ 要节约时间。例如对于一个有 100 个学生的班级来说，他们的差别相当于 5000 次运算和不到 700 次运算的差别。

其实还可以做进一步改善，将试卷也进行排序，这样试卷和名册就是一一对应的关系了，登录成绩只需要 n 次运算。不过将试卷排序可能要比将姓名排序慢很多，因为手动排序总比不上计算机排序的效率高。另一方面，将试卷排序后，也便于日后进行其他操作（比如，查看某个学生的试卷）。所以我们需要判断是否值得在个问题上花费时间。

4.6.3 评估程序执行时间

优化程序是一项很辛苦的工作，如非必要，肯定不愿意去做。在很多情况下，用于优化程序所花费的时间比优化后程序节约的时间还多。所以在优化之前，需要先测算一下程序执行所需要的时间，这是人们愿意花在优化上的时间上限。

在 R 之中，使用 system.time() 来估算程序执行所需的时间，例如：

```
> X <- rnorm(100000)
> Y <- rnorm(100000)
> Z <- rep(NA, 100000)
> system.time({
+ for (i in 1:100000) {
+ Z[i] <- X[i] + Y[i]
+ }
+ })
```

```
user  system elapsed
0.87    0.00    0.88
```

在上例中，user 是指用于这项任务所花费的时间，system 是指系统花费在其他任务上的时间，elapsed 是指整个过程花费的时间。

用户还可以评估一下到底程序的哪一部分最值得优化。一条著名的经验规则是 90/10 规则：即程序中 10% 的代码花费了 90% 的计算时间。所以应该关注这 10% 的瓶颈代码。总而言之，如果能确定并优化这个瓶颈，就可以得到一个显著的速度提高，而不必在意那 90% 的问题。

许多软件平台提供了剖析工具，以帮助找到的运算瓶颈。这类分析工具一般监控程序运算并报告执行情况。在 R 中的分析工具是 Rprof() 函数，不过我们不在此讨论该函数的细节。

4.6.4 选择其他工具

虽然我们很喜欢它，但不得不要承认， R 并不是唯一的计算平台，而且也不是对所有任务来说都是最佳的工具。对速度要求较多时，可以使用编译语言，例如 C ， C++ ，或 Fortran 。 R 本身是用 C 和 Fortran 编写的，这就是为什么它运算向量这么快，因为大部分工作直接由编译代码完成了。

我们建议使用 R 来完成大部分的工作。但如果对运算速度不满意，需要识别程序瓶颈或者需要将程序转换为编译语言， R 对很多程序语言都有广泛的支持。不过这些技术细节也超出了本书的范围。

4.6.5 谨慎优化

著名的计算机科学家 Donald Knuth 曾经说过：``在编程中，过早的优化是一切（或至少是大多数）罪恶的根源。"[2]

前面已经强调过，优化是一项很困难的事，所以并不建议必须进行优化。最后给出一些有效编程的建议：

1. 代码正确。
2. 运算速度尽量快。
3. 优化时保持代码正确。

本章习题

1. 用一个函数表达下面的多项式：

   $P(x) = c_n x^{n-1} + c_{n-1} x^{n-2} + \cdots + c_2 x + c_1$

   设函数名为 directpoly() ，这个函数的变量包括 x 和系数向量，返回这个多项式的值。

2. 在前面的函数中，对于一个中等及较大的 n ，使用 Horner 法则可以使计算 x 的值效率更高。

   (a) 将 c_n 的值赋给 a_n ； $a_n \leftarrow c_n$ 。

   (b) 取 i 的值为： $i = n-1, \ldots, 1$; 然后使 $a_i = a_{i+1} x + c_i$ 。

   (c) 返回 a_i .(这就是计算所得的结果 $P(x)$)

   编写一个 R 函数，命名为 hornerpoly() ，使用 x 和多项式的系数为参数，并返回多项式的计算结果。确保当输入的 x 为向量时，返回的值也是一个向量。

3. 比较上面两种算法所需的时间。

   (a) 首先尝试下面的代码：

[2] Knuth, D. E. (1974) Computer programming as art. Commun. ACM 17(12), 671.

```
> system.time(directpoly(x=seq(-10, 10, length=5000000),
+ c(1, -2, 2, 3, 4, 6, 7)))
> system.time(horner(x=seq(-10, 10, length=5000000),
+ c(1, -2, 2, 3, 4, 6, 7)))
```

(b) 当多项式的系数较小时，上面的比较结果有什么变化？当多项式改用下面的式子时，结果又如何？
$P(x) = 2x^2 + 17x - 3$

4. 请使用牛顿法计算下面函数的根，精确到小数点后 7 位，设初始估算值为 2.9 ，请问计算过程需要耗费多少时间？

(a) $(x-3)e^{-x}$

(b) $(x^2 - 6x + 9)e^{-x}$

5. 重复上面的问题，使用两分法和初始区间 $[2.1, 3.1]$ 计算。

6. 比较冒泡排序法（例 4.13 ）和合并排序法（例 4.14 ）所需的时间。分别尝试向量长度为 10， 1000，10000 和 100000 几种情况（可以使用 rnorm() 来生成向量）。

第 五 章

仿真

很多统计过程依赖于求随机变量的期望值以及分布的分位数。例如：

- 在假设检验中，样本的 P 值被定义为：当零假设成立时，观察数据至少和手头样本有一样极端分布的概率。它是随机变量的期望值，当样本分布没那么极端时期望值为 0 ，超过这个极端分布时为 1 。
- 估计值的偏差定义为估计值减去真值后的期望值。
- 置信区间依赖于分布的分位数。例如 $(\overline{X}-\mu)/(s/\sqrt{n})$。

在一些较简单的情况下，可以求分位数的值，或者使用大样本来求近似值。然而，在其他情况下，需要用计算机仿真的方式来模拟。

本章首先介绍蒙特卡洛仿真方法，包括随机数（或者更合适的说法是伪随机数）产生的基本思路。读者能了解如何从几种基本的概率分布模拟随机变量。然后，介绍仿真过程有趣的应用：用于求积分。最后，我们谈一下拒绝抽样法和重点抽样法以及一些高级方法的应用。

5.1 蒙特卡洛仿真

蒙特卡洛法是最常用的计算机模拟随机变量的方法。

为了得到均值 $\mu = E(X)$ 的近似值，蒙特卡洛方法首先产生 m 个独立同分布（i.i.d）的样本 X ，称为 $X_1, X_2, \cdots, X_m$ ，然后使用相同的均值 $\overline{X} = (1/m)\sum X_i$ 作为其期望值 $E(X)$ 。当 m 的值很大时， $\overline{X}$ 就是 $E(X)$ 很好的近似[1]。

此外，如果 m 的值很大，样本均值 $\overline{X}$ 的分布可以近似地表达为均值为 μ ，方差为 σ^2/m 的正态分布[2]。其中 σ^2 是 X 的方差，它可以用样本的方差 $s^2 = [1/(m-1)]\sum(X-\overline{X})^2$ 来近似表达。利用这个性质，可以构建均值 μ 的置信区间，例如 $\overline{X} \pm 1.96s/\sqrt{m}$ 就在 95% 的时间内包含了均值 μ 。

本章随后将讲述用电脑产生随机变量的仿真方法。用一些确定的方法产生一些数值，并将它们看做是随机数。在这个过程中，需要考虑两个有趣的参与者，首先是在幕后的程序员，他知道产生这些数值的算法是确定的可预测的；其次是数据的使用者，他们不在乎产生这些数据的机制。所以用户将这些数据看做随机数，不可预测。在实践中，两个参与者完全可以是同一个人。为了与真正的不可预测的随机数相区别，我们将用这种方法产生的数称为伪随机数。

5.2 生成伪随机数

首先简单介绍一下产生伪随机数的机制。这里描述一种最简单的方式，它用来产生位于 $[0,1]$ 区间内独立的均匀分布的随机变量。

使用一个乘法同余随机数发生器可以产生一系列伪随机数， $u_0, u_1, u_2, \cdots$, 它们看起来是处于 [0□1] 之间独立均匀分布的随机变量。

设 m 是一个很大的整数， b 是比 m 小的另一个整数。 b 的值通常选择接近于 m 的平方根。不同的 b 值和 m 值给伪随机数发生器不同的质量。有很多标准可用于判断如何选择这两个参数，但是最重要的还是要看产生的随机数是否合理。

首先选择一个处于 1 和 m 之间的整数 x_0 ，它被称作种子。后面会讨论选择 x_0 的策略。

$x_1 = bx_0 \text{ (mod m)}$

$u_1 = x_1/m.$

u_1 就是第一个位于 0 和 1 之间的伪随机数，第二个伪随机数也用同样的方法产生：

$x_2 = bx_1 \text{ (mod m)}$

$u_2 = x_2/m.$

[1] 这满足大数定理

[2] 这是中心极限定理

u_2 是另一个伪随机数。如果 m 和 b 的值选择得合适并且没有披露给用户，那么仅仅依靠 u_1 的值很难预测出 u_2 的。换句话说，对于大多数情况来说， u_2 可近似看做独立于 u_1 。随机数的产生过程根据下面的公式持续下去：

$x_n = bx_{n-1}$ (mod m)

$u_n = x_n/m$.

使用这种方法产生的数据完全是确定性的，但对于那些不了解上面公式的用户来说，这些数字的产生看起来就是随机的，不可预测的，至少在短期内可以这么说。

例 5.1

设 $m = 7$ 和 $b = 3$, 并且设 $x_0 = 2$ ，那么：

$x_1 = 3 \times 2$ (mod 7) = 6, $u_1 = 0.857$

$x_2 = 3 \times 6$ (mod 7) = 4, $u_2 = 0.571$

$x_3 = 3 \times 4$ (mod 7) = 5, $u_3 = 0.714$

$x_4 = 3 \times 5$ (mod 7) = 1, $u_4 = 0.143$

$x_5 = 3 \times 1$ (mod 7) = 3, $u_5 = 0.429$

$x_6 = 3 \times 3$ (mod 7) = 2, $u_6 = 0.286$.

必须注意，迭代过程将设 $x_7 = x_1$ ，然后重复前面的过程。所以相应地 u_i 也会循环。一个观察者可能不太容易从 u_1 来预测 u_2 ，但因为当 $i > 0$ 时， $u_{i+6} = u_i$ ，所以一个长的序列就比较容易被预测出来。为了产生不可预测的序列，需要设定一个非常大的循环长度，这样观察者不大容易看到整个循环过程。又因为循环长度不能超过 m 的值，所以选择 m 的时候就要选一个很大的值。

在选择 b 和 m 的时候一定要特别注意，确保循环的周期是 m 。假设有这样的情况，$b = 171$ ， $m = 29241$ ，初始 $x_0 = 3$ ，那么：

$x_1 = 171 \times 3 = 513$

$x_2 = 171 \times 513$ (mod 29241) = 0.

显然，随后所有的 x_n 都将是 0 。为了避免类似情况发生，我们选择 m 的时候要使得它不能被 b 整除，所以 m 最好是一个质数。下面一个例子是改进之后的的随机数发生器。

例 5.2

下面的命令产生 50 个伪随机数，它基于乘法同余的算法。

$x_n = 171x_{n-1}$ (mod 30269)

$u_n = x_n/30269$,

选择的初始值是 $x_0 = 27218$ 。

```
random.number <- numeric(50) # 用于存储生成的伪随机数
random.seed <- 27218
for (j in 1:50) {
+ random.seed <- (171 * random.seed) %% 30269
+ random.number[j] <- random.seed / 30269
+ }
random.number
```

存储在向量 *random.number* 中的结果如下所示。向量的元素都处于 0 和 1 之间，它们就是所谓 $u_1, u_2, \cdots, u_{50}$ 的伪随机数。

```
 [1] 0.76385080 0.61848756 0.76137302 0.19478675 0.30853348 0.75922561
 [7] 0.82757937 0.51607255 0.24840596 0.47741914 0.63867323 0.21312234
[13] 0.44391952 0.91023820 0.65073177 0.27513297 0.04773861 0.16330239
[19] 0.92470845 0.12514454 0.39971588 0.35141564 0.09207440 0.74472232
[25] 0.34751726 0.42545178 0.75225478 0.63556774 0.68208398 0.63636063
```

```
[31] 0.81766824 0.82126929 0.43704780 0.73517460 0.71485678 0.24051009
[37] 0.12722587 0.75562457 0.21180085 0.21794575 0.26872378 0.95176583
[43] 0.75195745 0.58472364 0.98774324 0.90409330 0.59995375 0.59209092
[49] 0.24754700 0.33053619
```

在 R 中有一个内置函数 runif() 具有类似的功能，不过它使用了不同的公式，有一个更长的循环周期。使用这个函数能自动产生伪随机数。

调用这个函数的语法：runif(n, min = a, max = b)

执行这个命令后产生 n 个位于区间 $[a,b]$ 之间均匀分布的伪随机数。 a 和 b 的默认值分别是 $a=0$，$b=1$。随机数种子由内部选定。

例 5.3

生成五个处于 $[0,1]$ 之间和 10 个处于 $[-3,-1]$ 之间，均匀分布的伪随机数。

```
runif(5)
```

```
[1] 0.9502223 0.3357378 0.1330718 0.4901114 0.0607455
```

```
runif(10, min = -3, max = -1)
```

```
[1] -2.284105 -2.545768 -2.199852 -1.126908 -1.324746 -2.744848
[7] -1.549739 -1.445740 -2.834744 -1.372574
```

读者执行上面的命令，得到的结果肯定会有所不同，这是因为生成随机数的种子不同。

选择 x_0 有两种策略。如果为了得到一个不可预测的序列，那么最好选择一个随机数。比如，电脑能感知内部时钟并精确到毫秒，所以可以用这一分钟过去了多少毫秒来作初始值。为避免被预测，这个外设的随机性只能使用一次，随后应该使用上面的公式来更新。例如，在上面的代码中，如果使用电脑时钟来产生每一个 u_i，那么对于一台很快的电脑来说，一毫秒就足以产生一个很长的序列。

选择 x_0 的第二种策略是使用一个固定值，例如 $x_0=1$。这样的好处是让 u_i 序列可预测可重复。这样选择对于包含随机数程序的调试很有好处，对于需要重复的过程也很有必要。在 R 中可使用 set.seed() 函数来实现。例如：

```
set.seed(32789) # 这确保读者能生成一样的随机数
runif(5)
```

```
[1] 0.3575211 0.3537589 0.2672321 0.9969302 0.1317401
```

习题

1. 使用下面的方式产生 20 个伪随机数。

   $x_n = 172x_{n-1} \bmod(30307)$

   选用初始种子 $x_0 = 17218$

2. 使用乘法同余发生器产生 20 个伪随机数，选用 $b = 171$ ， $m = 32767$ 。初始种子为 2018。
3. 使用 runif() 函数产生 10 个伪随机数，并设随机数种子为 set.seed(32078)。
   (a) 处于 $(0, 1)$ 之间的均匀分布。
   (b) 处于 $(3, 7)$ 之间的均匀分布。
   (c) 处于 $(-2, 2)$ 之间的均匀分布。
4. 使用 runif() 函数产生 1000 个伪随机数并保存为 U ，设种子为 set.seed(19908) 。
   (a) 计算 U 的均值，方差和标准差。
   (b) 与真实的均值，方差和标准差做比较。
   (c) 计算 U 之中小于 0.6 的数所占的比例。比较与真实值的差异。
   (d) 估计 $1/(U + 1)$ 的期望值。
   (e) 绘制 U 和 $1/(U + 1)$ 的直方图。
5. 产生 10000 个位于 $(3.7, 5.8)$ 之间均匀分布的随机数。
   (a) 估算这个随机变量的均值，方差和标准差，比较这个估算值和真值的差异。
   (b) 估算这个随机变量大于 4 的概率，并比较估算值和真值的差异。
6. 使用 runif() 分别模拟 10000 个处于 $(0, 1)$ 之间均匀分布的随机变量 U_1 和 U_2 ，并将它们存储为向量 $U1$ 和 $U2$ 。因为 $U1$ 和 $U2$ 近似于独立，所以可以将 U_1 和 U_2 看做独立的随机变量。
   (a) 估算期望值 $E[U_1 + U_2]$ ，并与真实值做比较。它与用 $E[U_1] + E[U_2]$ 的结果有何区别？
   (b) 估算 $Var(U_1 + U_2)$ 和 $Var(U_1) + Var(U_2)$ ，它们两个相等吗？两者的真实值会相等吗？
   (c) 估算概率 $P((U_1 + U_2) \leq 1.5)$ 。
   (d) 估算概率 $P((\sqrt{U_1} + \sqrt{U_2}) \leq 1.5)$ 。
7. 假定 U_1 ， U_2 和 U_3 是在 $(0, 1)$ 之间均匀分布的随机变量，使用仿真的方法估算下面的值：
   (a) $E[U_1 + U_2 + U_3]$
   (b) $Var(U_1 + U_2 + U_3)$ 和 $Var(U_1) + Var(U_2) + Var(U_3)$。
   (c) $E[\sqrt{U_1 + U_2 + U_3}]$
   (d) $P((\sqrt{U_1} + \sqrt{U_2} + \sqrt{U_3}) \geq 0.8)$。
8. 使用 round() 和 runif() 函数产生 1000 个 1 和 10 之间的伪随机数并将它们保存在 $discreteunif$ 变量中。使用 table() 函数检查每个值的频数是否与期望值相接近。如果不接近的话，有什么办法可改进？
9. 使用 sample() 函数能从一个向量空间中随机抽样。例如 sample(c(3,5,7),size = 2, replace = FALSE) 就是从向量 $(3, 5, 7)$ 之中不放回的随机抽两个样本。请使用 sample() 函数从 1 到 100 之间随机抽 50 个数，产生一个伪随机序列。
   (a) 采用有放回的抽样法。
   (b) 采用无放回的抽样法。
10. 下面的代码模拟两个均匀分布随机变量 U_1 和 U_2 的和（ X ）与差（ Y ），然后绘制 X 和 Y 的散点图，并估算两者的相关性。

```
U2 <- runif(1000)
U1 <- runif(1000)
X  <- U1 + U2
Y  <- U1 - U2
plot(Y ~ X)
cor(X,Y) # 计算样本相关性
```

两者的相关系数用于表达两个随机变量之间是否有线性相关。如果相关系数是 0，则表明两者几乎没有相关性。如果相关系数是 -1 或者 1，那么表示两者之间有很强的线性关系。执行上面的代码，并回答下面的问题。

(a) X 和 Y 之间是线性独立的吗？

(b) X 和 Y 之间是随机独立的吗？（回答该问题之前，请首先看两者的散点图。）

(c) U_1 和 U_2 之间是线性独立的吗？请用合适的方式检验。

(d) U_1 和 U_2 之间是随机独立的吗？用合适的图形来检验。

5.3 其他随机变量的仿真

5.3.1 伯努利随机变量

伯努利试验是一种只有两种可能结果的试验。例如，电灯泡要么亮要么不亮，两者必居其一。每一种结果（``亮" 或者 ``不亮"）都有与之相关联的可能性，而这两个可能性的和一定是 1。

例 5.4

设有一个学生在猜有五个选项问题的正确答案，所以他的正确率是 0.2，而不正确率则是 0.8。（学生的答案要么对，要么错，必居其一。）

假设想了解这个学生做 20 道题的成绩，可以用下面的思路来做仿真。

每个问题对应于一个独立的伯努利试验，其成功的概率等于 0.2。对于每一个问题生成独立的均匀分布随机数，用它来模拟学生答对问题的概率。如果这个生成的随机数小于 0.2，那么就说这个学生答对了，反之，这个学生就答错了。

用这种方法的原因在于，均匀分布随机变量小于 0.2 的概率恰好就等于 0.2；而大于 0.2 的概率则恰好是 0.8，而这正是学生答错的概率。这样可以用均匀分布的随机变量来模拟学生的行为。在 R 中可以用下面的代码实现：

```
set.seed(23207)  # 获得一致的结果
guesses <- runif(20)
correct.answers <- (guesses < 0.2)
correct.answers
```

```
 [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE TRUE TRUE FALSE
[12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE
```

correct.answers 是一个逻辑向量，它给出了所模拟学生的答案， TRUE 表示学生答案正确， FALSE 表示答案错误。我们可以据此计算正确答案的数量。

```
table(correct.answers)
```

```
correct.answers
FALSE  TRUE
   14     6
```

根据仿真的结果，该学生的成绩是 6/20。

在前面的例子中，可以用 “1” 和 “0” 分别代表伯努利试验的结果。以此定义伯努利随机变量：随机变量的值为 1 的概率为 p ，为 0 的概率为 $1-p$ 。

伯努利随机变量的期望值等于 p ，它的理论方差等于 $p(1-p)$ 。在上面的例子中，学生期望的正确率应该是 20% ，我们仿真的结果稍微要好一点，达到了 30% 。

习题

1. 写一个 R 函数对学生回答 n 个问题时猜对错的过程进行仿真。
   (a) 用这个函数模拟学生回答 10 个问题的过程，计算正确率。
   (b) 用这个函数来模拟学生回答 1000 个问题的过程，计算正确率。
2. 假设一个班级有 100 个学生，他们要回答 20 道对错选择题，每个学生都来猜正确答案。
   (a) 用仿真的方式来估算这个班的平均成绩，同时给出方差。
   (b) 估算成绩高于 30% 的学生所占的比例。
3. 编写一个 R 函数用来模拟 500 个灯泡，其中每一个有 99% 的概率是好的。使用仿真的方式估算随机变量 X 的期望值和方差，其中 1 表示灯泡是好的， 0 表示灯泡不好。这两个数字的理论值又分别是多少？
4. 编写一个 R 函数来模拟一个二项分布的随机变量，其中 $n=25$ ， $p=0.4$ （这相当于二十五个独立的伯努利随机变量 p 的和）。生成 100 个这样的随机变量，估算它的均值和方差，并与理论值（10，6）做比较。

5.3.2 二项分布随机变量

设 X 表示 m 个独立的伯努利随机变量的和，其中每个伯努利随机变量的概率为 p 。这样 X 就称作一个二项分布的随机变量，它表示在 m 次伯努利试验的成功数量。

一个二项分布随机变量的值可以是集合 $\{0,1,2,\cdots,m\}$ 中的一个数。随机变量 X 的值等于集合中某个数的概率符合二项分布。

$P(X=x)=\binom{m}{x}p^x(1-p)^{m-x}, x=0,1,2,\cdots,m$

这个概率值可以使用 dbinom() 函数来获得。语法为：dbinom(x, size, prob)。

这里， $size$ 和 $prob$ 分别表示二项分布中的参数 m 和 p ，而 x 表示 ``成功" 的次数。这个函数返回的值是 $p(X=x)$ 。

例 5.5

假设抛一个平整的硬币 6 次，计算有 4 次头像朝上的概率。

```
dbinom(x = 4, size = 6, prob = 0.5)
```

```
[1] 0.234375
```

这样 $P(X=4)=0.234$ ，其中 X 是一个参数为 $m=6$ 和 $p=0.5$ 的二项随机变量。

可以使用函数 pbinom() 来计算 $P(X\leq x)$ 的累计概率，这个函数的参数与 dbinom() 函数相同。例如，可以计算抛 6 次硬币，其中头像朝上的次数不超过 4 次的概率 $P(X\leq 4)$ ：

```
pbinom(4, 6, 0.5)
```

```
[1] 0.890625
```

函数 qbinom() 计算的则是二项分布的分位数。例如前面定义的随机变量 X， 在 89% 分位数处的分布是：

```
qbinom(0.89, 6, 0.5)
```

```
[1] 4
```

一个二项分布随机变量的期望值是 mp ，它的方差是 $mp(1-p)$ 。
可以使用 rbinom() 函数来生成二项分布伪随机数。其语法为:rbinom(n, size, prob)
其中 $size$ 和 $prob$ 是二项分布的参数， n 是希望生成的随机数数量。

例 5.6

假定某台机器每小时生产 15 个真空管，其中 10% 有缺陷。每个真空管的质量独立，即与其他真空管的质量无关。假如在一小时内检测出有缺陷产品超过 4 个，则说明生产过程失去了控制。模拟这台机器在 24 小时内每小时生产的缺陷产品个数，检验一下是否会在某些时刻对生产质量失去控制。

因为每小时生产 15 个真空管，其中 0.1 的可能性会产生缺陷，而且每个管子的质量独立。那么每小时产生缺陷品的数目是二项随机变量，参数是 $m = 15$ 和 $p = 0.1$ 。要模拟 24 小时内的生产，需要生成 24 个随机数，然后看这些随机数中有没有哪一个超过 5 。程序及结果如下：

```
defectives <- rbinom(24, 15, 0.1)
defectives
```

```
[1] 0 1 1 0 1 1 2 5 0 0 1 1 3 3 0 2 2 0 1 0 1 1 4 2
```

```
any(defectives > 5)
```

```
[1] FALSE
```

习题

1. 假设某工人每小时生产 25 件产品，其中缺陷率是 0.15 。模拟 24 小时内生产的缺陷产品数，看其中有没有超过 5 件缺陷产品的情况。分别重新计算缺陷率为 0.2 和 0.25 的情况。
2. 生成 10000 个二项分布的伪随机数，设其参数为 20 和 0.3 ，将这些生成的随机数赋值给变量 $binsim$ 。假设 X 是一个参数为 $(20, 0.3)$ 的随机变量。用仿真生成的随机数来估计下列值：
   (a) $P(X \leq 5)$
   (b) $P(X = 5)$
   (c) $E[X]$
   (d) $Var(X)$
   (e) 位于 95% 分位数处的 X ，（可以使用 quantile() 函数）。
   (f) 位于 99% 分位数处的 X 。

(g) 位于 99.999% 分位数处的 X 。

在每个问题中都比较估计值和真值之间的差别。假如要精确估计分位数，需要什么样的条件？

3. 用仿真的方法估算一个二项分布随机变量的均值和方差，设 $n = 18$ ， $p = 0.76$ 。请比较估算值和理论值的差异。

4. 下面这个函数使用所谓倒置 (inversion) 方法生成二项分布伪随机数。

```
ranbin <- function(n, size, prob) {
+     cumpois <- pbinom(0:(size - 1), size, prob)
+     singlenumber <- function() {
+         x <- runif(1)
+         N <- sum(x > cumpois)
+         N
+     }
+     replicate(n, singlenumber())
+ }
```

(a) 仔细阅读这个函数，写出说明文件。函数 singlenumber() 执行的是什么操作。[3]

(b) 使用 ranbin() 函数生成长度分别为 1000 , 10000 和 100000 的二项分布向量，分别使用 10 和 0.4 的参数。使用 system.time() 函数比较与使用 rbinom() 执行相应操作时所需时间的差异。

5. 下面的函数将相应的伯努利随机变量相加的方式生成二项分布伪随机数。

```
ranbin2 <- function(n, size, prob) {
+    singlenumber <- function(size, prob) {
+        x <- runif(size)
+        N <- sum(x < prob)
+        N
+        }
+    replicate(n,  singlenumber(size,  prob))
+ }
```

(a) 研究一下这个函数，并写出说明文档。特别注意一下 singlenumber() 函数的作用是什么。

(b) 使用 ranbin2() 函数生成 10000 个二项分布的随机数，设 *size* 参数分别为 10, 100 和 1000 ，*probability* 参数为 0.4 。比较使用 system.time() 函数与使用 rbinom() 执行相应操作时所需时间的差异。与上一题中使用 ranbin() 函数时的执行时间进行比较。

6. 前面这个随机数发生器要求，对于每生成一个二项分布，都需要生成 size 个均匀分布的伪随机数。下面这个发生器和前面那个的原理相同，但只需要生成一次均匀分布的随机数。

```
ranbin3 <- function(n, size, prob) {
+    singlenumber <- function(size, prob) {
+        k <- 0
+        U <- runif(1)
+        X <- numeric(size)
+        while (k < size) {
+            k <- k + 1
+            if (U <= prob) {
+                X[k] <- 1
+                U <- U / prob
+            } else {
```

[3] 使用 replicate() 函数，能重复调用 singlenumber() 函数，并把 n 个结果赋值给向量。具体可参阅 help(replicate) 的帮助文件。

```
+                 X[k] <- 0
+                 U <- (U - prob)/(1 - prob)
+             }
+         }
+         return(sum(X))
+     }
+     replicate(n, singlenumber(size, prob))
+ }
```

(a) 用 ranbin3() 函数，依据下面的参数生成 100 个二项分布的伪随机数。

(i) $size = 20$，$prob = 0.4$。

(ii) $size = 500$，$prob = 0.7$。

(b) 已知 $U < p$，那么 U/p 的条件分布是什么？

(c) 已知 $U < p$，那么 $(U - p)/(1 - p)$ 的条件分布是什么？

(d) 使用上面问题的答案，给出 ranbin3() 函数的说明文件。

7. 如果一个二项分布随机变量 X 具有参数 m 和 p，并且：

$$Z = \frac{X - mp}{\sqrt{mp(1-p)}}$$

那么根据中心极限定理， Z 就近似是标准正态分布，随着 m 的提高，近似度也提高。下面的代码按 m 从 1 到 100 生成随机数 Z，并绘出每个随机数的 QQ 图。

```
for (m in 1:100) {
+ z <- (rbinom(20000,size=m,prob=0.4)-m*0.4)/sqrt(m*0.4*0.6)
+ qqnorm(z,ylim=c(-4,4),main=paste("QQ-plot ,m=",m))
+ qqline(z)
+ }
```

(a) 执行上面的代码，看 Z 的分布如何随着 m 的值而变化。

(b) 修改上面的代码，显示当 p 分别为 $p = 0.3, 0.2, 0.1, 0.05$ 时分布的变化。 m 的值必须达到多大的时候才能使得 QQ 图的线看起来像直线。 $m = 100$ 足够大了么？

5.3.3 泊松随机变量

泊松分布是具有参数 n 和 p_n 的二项分布，其中 n 趋向于无穷而 p_n 趋向于 0，但是 np_n 的期望值收敛于一个常数 λ，方差 $np_n(1 - p_n)$ 收敛于相同的常数。这样泊松分布随机变量的均值和方差都等于 λ。这个参数有时候也被称做速率。

泊松随机变量来自许多不同方面。它们通常用来近似表示计数数据模型。例如某一地区在给定时间内发生地震的次数；或者一定时间内打银行客服的电话量。当把时间区间划分为 n 个阶段后，每个阶段发生事件的次数要么是零要么是一。泊松随机变量是事件发生的总次数，一个泊松随机变量 X 可能的取值是非负的整数 $(0, 1, 2, \cdots)$。取其中任何一个数的概率是：

$$P(X = x) = \frac{e^{-x}\lambda^x}{x!},\ x = 0, 1, 2, \ldots$$

在 R 中可以使用 dpois() 函数来获取概率的值。使用方法为：dpois(x, lambda)。其中 *lambda* 是泊松速率参数， x 是泊松事件发生的值。函数返回的值是 $P(X = x)$。

例 5.7

假定平均每分钟打进银行客服的电话是 0.5 个，那么根据泊松模型，下一分钟打进三个电话的概率是：

```
dpois(x = 3, lambda = 0.5)
```

```
[1] 0.01263606
```

所以，当 X 是泊松随机变量，并且均值为 0.5 时， $P(X = 3) = 0.0126$ 。
要计算 $P(X \leq x)$ 的累计概率函数，可以使用 ppois() 函数。泊松分布的分位数可以使用 qpois() 函数。
要生成泊松分布的随机数，可以使用 rpois() 函数。方法为：rpois(n, lambda)，其中参数 n 是生成随机数的数量， $lambda$ 的含义和前面相同。

例 5.8

假设某个十字路口平均每年发生交通事故 3.7 次。使用泊松模型模拟这个路口 10 年的车祸数量。

```
rpois(10, 3.7)
```

```
[1] 6 7 2 3 5 7 6 2 4 4
```

泊松过程
泊松过程是用来表达某一时间段内发生事件的简单模型。考虑泊松过程的一个角度可以是从一条线或一个平面随机抽取``点"（如有必要，也可以从更高维度的几何体中抽取）。
齐次的泊松过程有下列的特点：

1. 在集合内包含点的数目具有泊松分布，这个分布的速率参数正比于集合的大小。
2. 在不交叉的集合中包含点的数目是彼此独立的。

在实际中，假如一个泊松过程有参数 λ ，那么在区间 $[0, T]$ 之间的点就具有均值为 λT 的泊松分布。要做此模拟的话，可以用下面的步骤：

1. 生成 N 个参数为 λT 的泊松分布伪随机数。
2. 在区间 $[0, T]$ 内生成 n 个独立的均匀分布的伪随机数。

例 5.9

在区间 $[0, 10]$ 上，以速率参数 1.5 模拟齐次的泊松过程。

```
N <- rpois(1, 1.5 * 10)
P <- runif(N, max = 10)
sort(P)
```

```
[1] 0.03214420 0.11731867 0.25422972 2.43762063 3.93583254 4.05037783
[7] 4.50931123 5.28833876 7.43922932 8.47007125 8.76151095 8.81578295
[13] 9.35800644
```

习题

1. 假定平均每年发生车祸 2.8 次，请用泊松模型模拟 15 年内每年发生车祸的数量。
2. 假设每辆轿车表面有 1.2 个瑕疵，请模拟 20 辆车总共有多少表面瑕疵。
3. 设一个泊松随机变量的均值是 7.2，生成 10000 个伪随机数后，比较随机数的均值、方差与理论值的差异。
4. 分别以均值为 $5, 10, 15$ 和 20 生成 10000 个伪随机数，并将它们保存在变量 $P5, P10, P15$ 和 $P20$ 中。
   (a) 估算 $\mathrm{E}[\sqrt{X}]$ 和 $\mathrm{Var}(\sqrt{X})$, 其中 X 是参数分别为 $\lambda = 5$, 10, 15, 和 20 的泊松分布。
   (b) 因为当 X 是泊松分布时，它的方差随着均值同步增加。那么对 X 取平方根之后，方差和均值的关系怎么变化呢？（统计学家经常通过对计数数据取平方根的方法来“稳定方差”，您能理解这种做法吗？）。
5. 用一个仿真试验来验证下面这样的命题：一个参数是 1.5 的泊松过程，它落在区间 $[4, 5]$ 之间点的数目，也服从均值为 1.5 的泊松分布。首先在区间 $[0, 10]$ 之间模拟大量的泊松过程，然后计算落在 $[4, 5]$ 之间点的数量。把获得的点的数量分布和相应的泊松分布用 QQ 图作比较。下面提供了这个过程的代码，读者可以执行并查看结果。

```
poissonproc <- function() {
+ N <- rpois(1, 1.5 * 10)
+ P <- runif(N, max = 10)
+ return(sum( 4 <= P & P <= 5 ))
+ }
counts <- replicate(10000, poissonproc())
qqplot(counts, rpois(10000, 1.5))
abline(0, 1) # 点的位置很靠近这条线
```

6. 中心极限定理有这样一种描述方法：如果 X 是参数为 λ 的泊松随机变量，并且

$$Z = \frac{X - \lambda}{\sqrt{\lambda}}$$

   那么 Z 近似服从标准正态分布，并且随着 λ 的增加，近似度也提高。下面的代码在 λ 分别在 $\{1, 3, \ldots, 99\}$ 时模拟了大量的 Z 值，然后与正态分布用 QQ 图做比较。

```
for (m in seq(1, 120, 2)) {
+ z <- (rpois(20000, lambda = m) - m) / sqrt(m)
+ qqnorm(z, ylim = c(-4, 4), main = "QQ-plot")
+ qqline(z)
+ mtext(bquote(lambda == .(m)), 3)   # 生成副标题
+ }
```

   (a) 执行上面的代码，观察 Z 的分布如何随着 λ 的增加而变化。
   (b) 当 λ 增加到多大时，QQ 图中的点才与直线比较吻合？
7. 在区间 $[0, 2]$ 之间，用参数 2.5 进行 10000 次泊松仿真过程。
   (a) 每一次仿真过程中，正好落在区间 $[0, 1]$ 和 $[1, 2]$ 之间的点有多少个。
   (b) 落入这两个区间中的点的数量符合参数为 2.5 的泊松分布吗？
   (c) 能不能假设落在 $[0, 1]$ 之间的点的数量与落在 $[1, 2]$ 之间的点的数量没有关联？请用合适的散点图来证明。（提示，需要用 jitter() 函数）

5.3.4 指数分布随机变量

指数分布随机变量被用作模拟诸如机械或电子组件失效的时间, 也可以模拟一个服务员完成一次客户服务所需的时间。描述一个指数分布可以用表示故障率的常数 λ 表示。

如果 $\lambda > 0$，那么对于任何非负的 t，满足 $P(T \leq t) = 1 - e^{-\lambda t}$ 的 T 具有指数分布。在 R 中，pexp() 函数能表征这个分布。调用方式为：pexp(q, rate)，函数的输出是 $P(T \leq q)$，其中 T 是参数为 $rate$ 的指数分布随机变量。

例 5.10

假设银行职员给每个客户服务的时间可以用指数分布随机变量来模拟，平均每个客户的服务时间是 3 分钟。那么一个客户接受服务的时间小于一分钟的概率为：

```
pexp(1,rate=3)
```

```
[1] 0.950213
```

即当 X 满足 $rate$ 为 3 的指数分布时，$P(X \leq 1) = 0.95$。

将分布函数右侧对 t 求微分就得到概率密度函数：

$f(t) = \lambda e^{-\lambda t}$

在 R 中用 dexp() 函数来计算概率密度，它的参数和 pexp() 函数一样。使用 qexp() 函数可以得到指数分布的分位数。

指数分布随机变量的期望值等于 $1/\lambda$，方差等于 $1/\lambda^2$。

可以使用反函数法生成指数分布的伪随机数。对于一个指数分布的随机变量 $F(x) = 1 - e^{-\lambda x}$，它的反函数是 $F^{-1}(x) = -\frac{\log(1-U)}{\lambda}$。当任何 $x \in (0,1)$，都有

$P(F(T) \leq x) = P(T \leq F^{-1}(x)) = F(F^{-1}(x)) = x.$

这样，$F(T)$ 就是在区间 $0,1$ 之间均匀分布的随机变量。前面已经知道如何生成均匀分布的伪随机数，那么通过反函数转换 $F^{-1}(x)$ 就可以生成指数分布的随机变量。即，首先生成在区间 $[0,1]$ 上均匀分布的伪随机数 U，然后根据 $1 - e^{-\lambda T} = U$ 解出 T，$T = -\frac{\log(1-U)}{\lambda}$。此时 T 就满足参数为 λ 的指数分布。

在 R 中，用函数 rexp() 可以生成 n 个指数分布变量。命令为：rexp(n, rate)。

例 5.11

银行出纳员需要对 10 个正在排队的客户服务。他给每个客户服务的时间是参数 $rate$ 为 3 分钟的指数分布，模拟他给每个客户提供服务的时间。

```
servicetimes <- rexp(10, rate = 3)
servicetimes
```

```
[1] 0.25415279 0.79177402 0.24280817 0.07887371 0.10738250
[6] 0.83294959 0.09676131 0.16938459 0.53317718 0.16583246
```

他给这 10 个客户提供服务的总时间大约是三分钟零十六秒。

```
sum(servicetimes)
```

```
[1] 3.273096
```

另一个模拟泊松过程的方法

可以证明，参数为 λ 的齐次泊松过程在线上的点，被独立的指数分布随机变量所分割，这个指数分布随机变量的均值恰好是 $1/\lambda$。用这个原理，可以构建另一种简单的模拟泊松过程的方法。

例 5.12

从 0 开始，按 rate = 1.5 生成前 25 个点。

```
X <- rexp(25, rate = 1.5)
cumsum(X)
```

```
 [1]  0.03393865  2.20285768  3.19988695  5.46515631
 [5]  5.78858794  6.22367839  6.30169938  7.83505942
 [9]  9.44275687  9.69396031 11.26030185 12.11176412
[13] 12.22637985 12.70107374 12.77394543 17.46457220
[17] 17.82666813 17.94375221 18.40288641 18.73823627
[21] 19.64013050 19.78668741 19.93451087 21.92672587
[25] 23.62520573
```

习题

1. 模拟 50000 个参数为 3 的指数分布随机数。
   (a) 这些数中，小于 1 的数占多少百分比。与理论值做比较。
   (b) 它们的平均值是多少？理论值是多少？
   (c) 这个样本的方差是多少？与理论值做比较。
2. 假设某品牌电池的寿命符合均值为 55 小时的指数分布，用仿真方法估算这种电池的平均寿命和方差，并与理论值相比较。
3. 一个电子设备包含两个元件，每个元件的寿命都可以用独立的指数随机变量来模拟，第一个元件的平均失效时间是三个月，第二个元件是六个月。假如两个元件中的任何一个失效，该电子设备就失效。使用仿真方法来估算这个电子设备的平均使用寿命及其方差。
4. 假如在上一题中，只有当两个元件都失效时，电子设备才失效。那么它的平均寿命和方差又是多少？
5. 用本节描述的方法，模拟 10000 次参数为 2.5 的泊松过程。检验位于区间 $[0, 2]$ 之间的点，这些点的数量应该接近均值为 5 的泊松分布。

5.3.5 正态分布随机变量

如果一个随机变量 X 的概率密度函数符合 $f(x) = \dfrac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$，那么 X 就是正态分布的随机变量，其中 μ 是 X 的期望值，σ^2 是 X 的方差。当 $\mu = 0$，且 $\sigma = 1$ 时，这个分布称为标准正态分布。

正态分布的密度函数可以用 dnorm() 来获得，分布函数则可以用 pnorm() 来获得，对应的分位数函数是 qnorm()。例如，一个均值为 2.7，标准差为 3.3 的正态分布，它的第 95% 的分位数是：

```
qnorm(0.95, mean = 2.7, sd = 3.3)
```

```
[1] 8.128017
```

在 R 中，可以使用 rnorm() 函数来产生正态分布的伪随机数。语法为：rnorm(n, mean,sd)。它将生成 n 个均值为 $mean$，标准差为 sd 的正态分布伪随机数。

例 5.13

可以用下面的代码生成 10 个均值为 -3，标准差为 0.5 的独立正态分布随机变量：

```
rnorm(10, -3, 0.5)
```

```
[1] -2.986856 -3.495919 -3.300159 -3.646522 -3.741575
[6] -3.287324 -3.527170 -2.576409 -3.062986 -2.772351
```

如果要生成满足某种条件分布的随机数，可以先生成非条件分布的随机数，然后剔除那些不满足条件的值。

例 5.14

从标准正态分布中生成随机变量 x，并使得 $0 < x < 3$。可以先从整个正态分布的区间中生成随机数，然后剔除那些不符合要求的结果。

```
x <- rnorm(100000) # 生成标准正态分布随机数
x <- x[(0 < x) & (x < 3)] # 剔除在 (0,3) 之外的值
hist(x, probability=TRUE) # 显示最终结果
```

图 5.1 显示了按这样的条件生成的随机变量在 $[0,3]$ 区间内的分布。

习题

1. 模拟 100 次均值为 51，标准差为 5.2 的正态分布随机变量，估算这些仿真结果的均值和方差，并与理论值做比较。
2. 模拟 1000 次标准正态分布随机变量 z，然后用仿真结果做下面的计算：
   (a) $P(Z > 2.5)$
   (b) $P(0 < Z < 1.645)$
   (c) $P(1.2 < Z < 1.45)$
   (d) $P(-1.2 < Z < 1.3)$.

   将上面的结果与理论值做比较。
3. 模拟均值为 3，方差为 16，并且 $|X| > 2$ 的正态分布随机变量的条件分布。

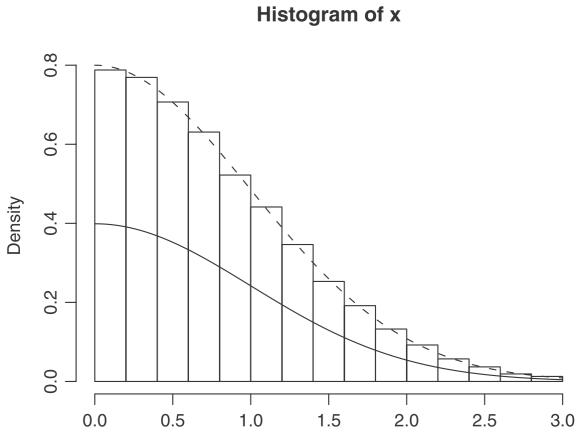


图 5.1: 在区间 $[0,3]$ 正态分布随机变量直方图及密度函数曲线

4. 已知自由度为 1 的 χ^2 的随机变量与标准正态分布的平方根有相同的分布。模拟 100 次这样的 χ^2 随机变量，估算它的均值和方差（并与理论值 1 和 2 做比较）。

5. 一个自由度为 n 的 χ^2 分布随机变量，与 n 个独立的标准正态分布随机变量之和有相同的分布。模拟自由度为 8 的 χ^2 随机变量，估算它的均值和方差，并与理论值 8 和 16 做比较。

5.4 蒙特卡洛积分

假设函数 $g(x)$ 在区间 $[a,b]$ 上可积。那么 $\int_a^b g(x)dx$ 给出了 x 位于 $[a,b]$ 之间，y 位于 0 和 $g(x)$ 之间的曲线所围的面积（如果 $g(x)$ 是负值，那么被视为负的面积）。

蒙特卡洛积分利用仿真方法来近似计算这些积分。它的计算原理是依赖大数定理，该定理认为大量随机抽样后，样本均值接近于样本分布的期望值。如果能将积分用期望值来表示，那么就能用样本均值来近似计算。

假设 $U_1, U_2, \ldots, U_n$ 是在区间 $[a,b]$ 上独立的均匀分布随机变量，它们都有同样的密度函数 $f(u) = 1/(b-a)$。那么它们的期望值为：

$$E[g(U_i)] = \int_a^b g(u)\frac{1}{b-a}du$$

所以原来的积分 $\int_a^b g(x)dx$ 就可以用 $b-a$ 乘上样本均值 $g(U_i)$ 来近似计算。

例 5.15

估算积分 $\int_0^1 x^4 dx$:

```
u <- runif(100000)
mean(u^4)
```

```
[1] 0.2005908
```

与它的真实结果 0.2 相比，使用这种方法显然更容易。

例 5.16

估算积分 $\int_2^5 sin(x)dx$:

```
u <- runif(100000, min = 2, max = 5)
mean(sin(u)) * (5-2)
```

[1] -0.6851379

它的真实结果应该是 -0.700。

多重积分

假定 $V_1, V_2, \ldots, V_n$ 是另一个在区间 $[0,1]$ 上独立的均匀分布随机变量，假设函数 $g(x,y)$ 在变量 x 和 y 上可积。根据大数定理

$$\lim_{n\to\infty}^{n} g(U_i, V_i)/n = \int_0^1 \int_0^1 g(x,y)dxdy$$

于是可以生成两个独立的均匀分布随机变量，通过它们来计算积分 $\int_0^1 \int_0^1 g(x,y)dxdy$，首先分别对某一个随机变量计算 $g(U_i, V_i)$，然后计算他们的均值。

例 5.17

用下面的程序代码估算积分 $\int_3^{10} \int_1^7 sin(x-y)dxdy$:

```
U <- runif(100000, min = 1, max = 7)
V <- runif(100000, min = 3, max = 10)
mean(sin(U - V))*42
```

[1] 0.07989664

其中，$42 = (7-1)(10-3)$，用于补偿 U 和 V 的联合密度函数 $f(u,v) = 1/42$。

均匀分布的密度函数并不是蒙特卡洛积分中唯一能用的密度函数。如果变量 X 的密度是 $f(x)$，那么 $E[g(X)/f(X)] = \int [g(x)/f(x)]f(x)dx = \int g(x)dx$，所以可以通过计算 $g(X)/f(x)$ 样本的均值来估算积分。

例 5.18

为了估算积分 $\int_1^\infty \exp(-x^2)$，可以先将函数改写成 $\int_0^\infty \exp[-(x+1)^2]$，然后生成指数分布的随机数 X：

```
X <- rexp(100000)
mean(exp(-(X+1)^2)/dexp(X))
```

0.1401120

这个积分的真实值是 0.1394.

蒙特卡洛积分并不总有效，有时候 $g(X)/f(X)$ 的比值变化很大，使得样本的均值并不收敛。这时候需要选择 $f(x)$ 使该比值能大致固定，避免 $g(x)/f(x)$ 太大。

习题

1. 使用蒙特卡洛法估算下面的积分，如果知道它们的真实值的话，比较两者差异。

   $\int_0^1 xdx$ $\int_1^3 x^2dx$ $\int_0^\pi sin(x)dx$ $\int_1^\pi e^x dx$

   $\int_0^\infty e^{-x}dx$ $\int_0^\infty e^{-x^3}dx$ $\int_0^3 sin(e^x)dx$ $\int_0^1 \frac{1}{\sqrt{2\pi}}e^{-x^2/2}dx$

   $\int_0^2 \frac{1}{\sqrt{2\pi}}e^{-x^2/2}dx$ $\int_0^3 \frac{1}{\sqrt{2\pi}}e^{-x^2/2}dx$

2. 用蒙特卡洛法计算双重积分

   $\int_0^1 \int_0^1 cos(x-y)dxdy$ $\int_0^1 \int_0^1 e^{-(x+y)^2}(x+y)^2dxdy$

   $\int_0^3 \int_0^1 cos(x-y)dxdy$ $\int_0^5 \int_0^2 e^{-(x+y)^2}(x+y)^2dxdy$

5.5 高级仿真方法

到目前为止所讲的仿真方法只能用于特定的概率分布函数。而更通用的仿真方法可以从大量不同的分布函数中抽取伪随机数。

例 5.19

假设 X 是参数为 (n,p) 的二项分布随机变量，其中 n 已知，但 p 的确切值不知道，只知道它大约在 0.7 附近。已知数据 $X=x$，我们需要来估算 p。

对 p 的极大似然估计是 $\hat{p}=x/n$，但这样做并没有用到已知的信息，即 p 大概等于 0.7。假如将已知信息转换成一个密度函数，即 $p\sim N(0.7,\sigma=0.1)$，那么利用贝叶斯公式就能在已知 $X=x$ 时估算 p 的条件密度：

$$f(p|X=x)\propto \exp\left(\frac{-(p-0.7)^2}{2(0.1)^2}p^x(1-p)^{n-x}\right), \text{其中 } 0<p<1$$

要通过计算得到这个函数的精确解很困难，但通过蒙特卡洛这样的数值估算方法则比较容易。

这一节主要介绍能用于计算上述例子的伪随机数生成机制，主要使用两种仿真方法：

- 拒绝抽样 (regection sampling)
- 重点抽样 (importance sampling)

5.5.1 拒绝抽样

拒绝抽样的原理已经应用在 5.3.5 节从条件分布中抽样的例子里：从一个合适的分布中抽样，然后二次抽样来获取目标分布。下面用两个例子来分别说明，从单变量密度函数或者概率函数 $g(x)$ 中用拒绝抽样法抽取随机样本。

第一个例子演示最简单的拒绝抽样。

例 5.20

从三角形密度函数中模拟伪随机数

$$g(x)=\begin{cases}1-|1-x|, & 0\le x<2\\ 0 & \text{otherwise}\end{cases}$$

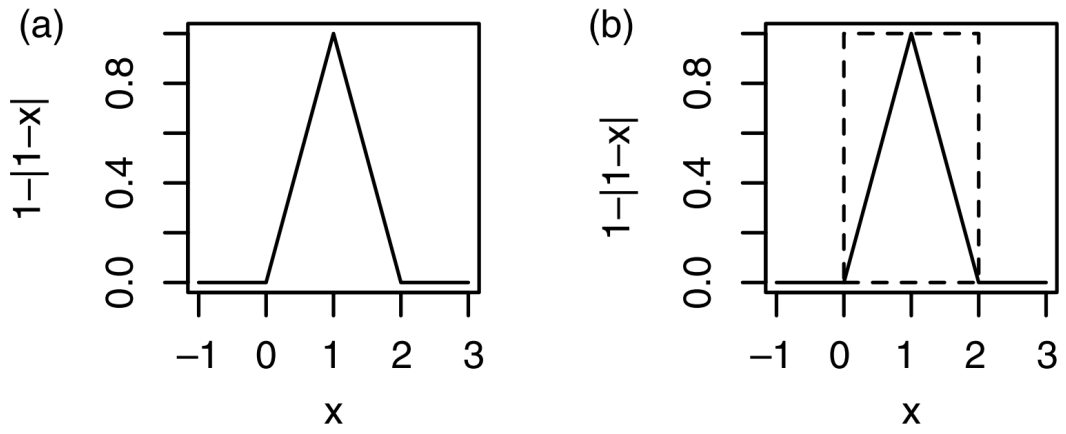


图 5.2: 三角密度函数

这个密度函数如图5.2(a) 所示。如果能均匀地从三角形区域抽取点，那么 x 轴上的分布就满足 $g(x)$ 的密度函数。图5.2(b) 显示，密度非零的区域可以完全包含在一个宽是 2，高是 1 的矩形中。在矩形区域均匀分布的点的子集，也会均匀分布在三角形区域中，它就对应了三角形密度分布。这样，仿真的步骤就是：

1. 在矩形区域中生成均匀分布的随机点 (U_1,U_2)。

2. 如果 (U_1, U_2) 位于三角形区域内，那么接受 U_1 为认可的伪随机数，否则就拒绝它，并返回到第一步。

因为三角形区域占据了矩形区域一半的面积，所以大概每生成两个矩形中的均匀分布点，会有一个能被接受在三角形分布区域内。

```
U1 <- runif(100000, max=2)
U2 <- runif(100000)
X <- U1[U2 < (1 - abs(1 - U1))]
```

向量 X 大约包含了 50000 个从三角形分布中模拟的值。

为了满足 $g(x)$ 可能在比较大的区间内（甚至在无限区间内）非零，同时也是为了提高计算效率，需要一个更通用的拒绝抽样方法。我们可以找到一个合适的常数 k，以及一个容易抽样的函数 $f(x)$，使得它们在所有的 x 上都满足 $kg(x) \le f(x)$。这样在 $f(x)$ 下均匀地抽取满足 $kg(x)$ 条件的点，就可以得到期望的分布，见图5.3。其中实线表示 $f(x)$ 的密度分布，虚线表示 $kg(x)$ 的密度分布。在 $f(x)$ 下均匀分布了仿真点，如果这些点位于虚线之上，则被拒绝，而位于虚线之下的则是被接受的仿真点。

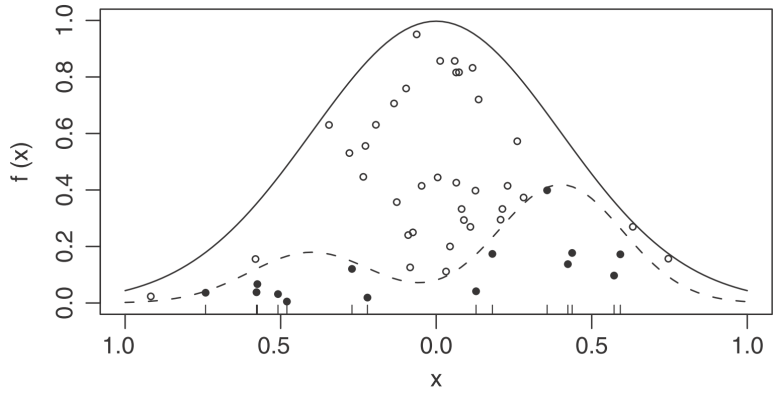


图 5.3: 拒绝抽样法

```
repeat {
draw X ~ f(X) , U ~ unif(0,1)
if U * f(X) < kg(X)
break
}
output X
```

例 5.21

从概率密度函数 $g(x) = Ce^{-x^{1.5}}$ 生成仿真数据，其中 C 是未知常数，目的是将整个概率密度函数归一化。即使不知道 C 取何值，但因为 $0.5e^{-x^{1.5}} \le e^{-x}$，可以选择 $k = 0.5/C$ 来进行拒绝抽样。

```
kg <- function(x) 0.5 * exp(-(x^1.5))
X <- rexp(100000)
U <- runif(100000)
X <- X[U * dexp(X) < kg(X)]
```

向量 X 中就包括了大量符合要求的伪随机数，可以进一步画这些随机数的分布：

```
hist(X, freq = FALSE, breaks="Scott")
```

此处在直方图中选用了相对频率是为了方便叠加理论概率密度曲线。但问题是不知道 C 来确定 $g(x)$ 的概率密度函数。我们可以通过蒙特卡洛积分求解，或者用更简单的方法，通过寻找合适的抽样比例 k。

```
k <- length(X)/100000
g <- function(x) kg(x)/k
curve(g, from=0, to=max(X), add=TRUE)
fbyk <- function(x) dexp(x)/k
curve(fbyk, from=0, to=max(X), add=TRUE, lty=2)
```

结果如图5.4所示。

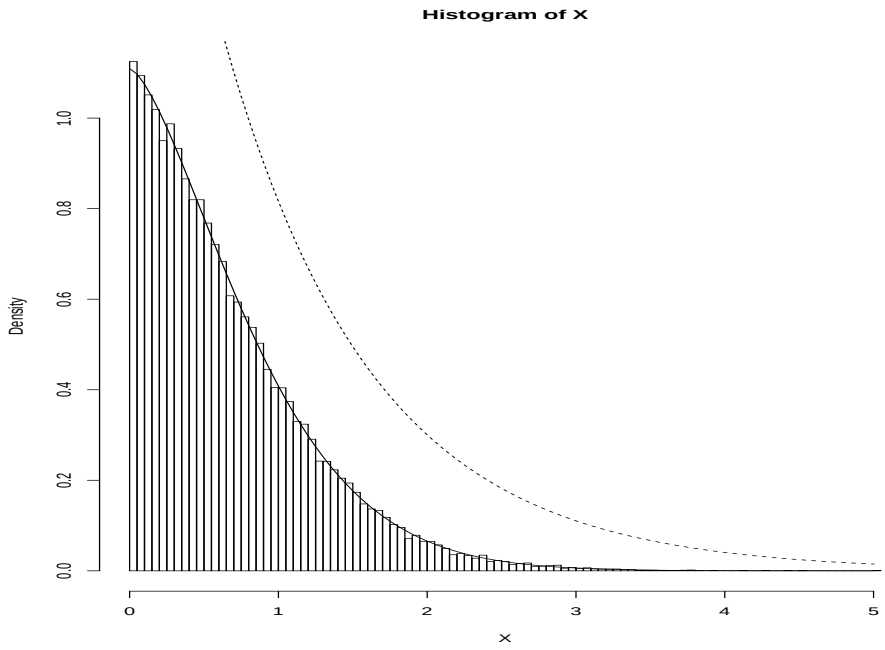


图 5.4: 拒绝抽样法 -实线为 $g(x)$，虚线为 $f(x)/k$。

5.5.2 重点抽样

加权平均值是对每个观察值加权，然后求平均值的方法。通常计算平均值的方法是：$\bar{x} = (1/n)\sum_{i=1}^{n} x_i$，加权平均值的计算方法是：$\bar{x}_w = \sum_{i=1}^{n} w_i x_i / \sum_{i=1}^{n} w_i$。

重点抽样法是在进行随机进行抽样时同时生成权重的技术，通过这种方法，使得加权平均值近似于目标概率密度函数 $g(x)$ 的期望值。和拒绝抽样一样，首先从便于抽样的函数 $f(x)$ 开始：

1. 选择一个知道如何抽样的，合适的概率密度函数 $f(x)$。

2. 从 $f(x)$ 中抽取样本 $x_1, x_2, \ldots, x_n$。

3. 计算权重 $w_i = g(x_i)/f(x_i)$。

现在可以用 $h(x_i)$ 的权重 w_i 来估算函数 $h(X)$ 的期望值，其中 $X \sim g(x)$。这个方法之所以可行，是因为权重 w_i 正比于拒绝抽样法中接受 x_i 来自于 $g(x)$ 的概率。给定 x_i，加权平均值计算公式中的系数 $w_i h(x_i)/\sum_{i=1}^{n}$，正是在拒绝抽样中抽取这个样本点相对与样本均值的权重。既然拒绝抽样有效，那么重点抽样就同样有效，而且效率更高，因为被拒绝的抽样点的信息并没有被丢弃。进一步来说，它没必要寻找拒绝抽样法中必要的常数 k，函数 $g(x_i)$ 和 $f(x_i)$ 也没必要归一化，因为除以权重之和后，函数自然有了恰当的比例。

不过，重点抽样法比简单随机抽样要困难得多。所以在很多情况下，人们更愿意使用拒绝抽样法。

下面紧接着上一节的例子来说明重点抽样法，使用 $weighted.mean$ 函数来估算函数 $g(x)$ 的均值和方差。

```
X <- rexp(100000)
W <- g(X)/dexp(X)
mean <- weighted.mean(X, W)
mean
```

[1] 0.6579574

```
weighted.mean((X - mean^)2, W)
```

[1] 0.3036045

习题

1. 利用 runif() 和拒绝抽样法，生成在区间 $[-4,4]$ 内的标准正态分布伪随机数。能进一步生成在整个实数轴上正态分布的伪随机数吗？

2. 下面的程序能生成正态分布伪随机数：

```
> rannorm <- function(n, mean = 0, sd = 1){
+ singlenumber <- function() {
+ repeat{
+ U <- runif(1)
+ U2 <- sign(runif(1, min = -1)) # 返回值是 +1 或 -1
+ Y <- rexp(1)*U2 # Y 是双重指数分布随机变量
+ if (U < dnorm(Y)/ exp(-abs(Y))) break
+ }
+ return(Y)
+ }
+ replicate(n, singlenumber())*sd + mean
+ }
```

   (a) 利用这个函数生成均值为 8，标准差为 2 的 10000 个正态分布伪随机数。

   (b) 用 *QQ* 图来检验这个函数的精确性。

   (c) 用函数 curve() 在区间 $[0,4]$ 之内画出标准正态分布的密度曲线。使用参数 add = TRUE 在图上叠加指数分布密度曲线，证明已经恰当地应用了拒绝抽样方法。

3. 分别用下面两种方法来模拟离散分布的随机变量，这个随机变量的取值分别为 $0,1,2,3,4,5$ 而取这些值的相应概率分别为 $0.2, 0.3, 0.1, 0.15, 0.05, 0.2.$。第一种方法是反函数法：

```
> probs <- c(0.2, 0.3, 0.1, 0.15, 0.05, 0.2)
> randiscrete1 <- function(n, probs) {
+ cumprobs <- cumsum(probs)
+ singlenumber <- function() {
+ x <- runif(1)
+ N <- sum(x > cumprobs)
+ N
+ }
+ replicate(n, singlenumber())
+ }
```

   第二种方法是拒绝抽样法：

```
> randiscrete2 <- function(n, probs) {
+ singlenumber <- function() {
+ repeat{
+ U <- runif(2, min=c(-0.5, 0), max=c(length(probs)-0.5,
+ max(probs)))
+ if (U[2] < probs[round(U[1])+1]) break
+ }
+ return(round(U[1]))
+ }
+ replicate(n, singlenumber())
+ }
```

运行上面两种方法，生成 $n = 100, 1000,$ 和 10000 个随机数，用 $system.time()$ 函数比较哪一种方法的运行速度更快。

4. 使用由下面的程序定义的在 $\{0, 1, 2, \ldots, 99\}$ 上的概率分布，重做上面的习题。

```
> set.seed(91626)
> probs <- runif(100)
> probs <- probs / sum(probs)
```

什么时候拒绝抽样法会比反函数法更有效？

5. 写一个函数，使用重点抽样法来生成参数为 (m, p) 的二项分布伪随机数。将加权平均值和理论值做比较。

本章习题

1. 写一个函数来模拟两个人（Ann 和 Bob）打乒乓球，假定第一个得到 21 分的人获胜。

   (a) 假设每个人击打到球的概率分别是 p_{Ann} 和 p_{Bob}，那么当两者的击球概率取不同的值时，Ann 最终获胜的概率是多少？

   (b) 可以增加更多的参量，比如发球能力，扣球能力和削球能力。读者可以根据想象自己添加。

2. 下面的模型可用于研究传染病。假设共有 N 个人，其中有一些人染病，并且：

   - 当一个病人遇到一个健康人后，后者有 α 的概率会被感染。
   - 所有的相遇都仅发生在两个人之间。
   - 任何两个人相遇的概率相等。
   - 在单位时间内，会有一次相遇。

   (a) 写一个函数来模拟这个模型，并用多个参数来运算，比如 N 取 10000，α 取 0.001 和 0.1 之间的值。假设模型开始时，有一个人感染了疾病，那么这个模型运行的过程会怎样？

   (b) 假设开始时有一个人染病，那么要用多长时间才能感染 1000 个人？

   (c) 假设被感染的人在单位时间内有 0.01 的概率会痊愈，重新检视这个模型的运行过程，并与前面的运行结果相比较。

   (d) 假设任何两个人相遇的时间可以用均值为 5 分钟的指数分布随机变量来模拟，那么上面的结果有何变化？

   (e) 假设任何两个人相遇的时间是均值为 5 分钟，标准差为 1 分钟的随机变量的绝对值。上面的结果有何变化？

3. 模拟下面这些汽车保险的情况：

- 保险索赔案每年的增长符合参数为 100 的泊松过程。
- 每一笔索赔额度符合形状参数和尺度参数都等于 2 的伽马分布。这个分布的均值为 1，方差为 $1/2$。保险公司应该尽快支付他们的索赔。
- 保险公司每年赚得的保费为 105，各年间平均分布。

写出 R 程序来计算下面问题：

(a) 模拟在一年内，索赔的次数和总金额。用图表示一年内保险公司持有金额的总数，它应该从零开始，按照保险费率平滑上升，每遇到一次索赔就下跌一次。

(b) 重复仿真过程 1000 次，估算下面的值：
(i) 保险公司期望能获取的最低收益。
(ii) 保险公司期望最后能获得的收益。

4. 设 $f(x) = (\sin x)^2$ for $0 < x < 2\pi$.

(a) 绘制函数图。

(b) 用蒙特卡洛积分法估算在区间 $0 < x < 2\pi$ 内函数曲线下的面积，以及估算值 95% 的置信区间。

(c) 用微积分计算这个面积的真实值，它位于刚才估算的置信区间内吗？

(d) 写一个函数 $rsin2$ 从密度函数 $f(x)/k$ 并在区间 $0 < x < 2\pi$ 内产生随机数（其中 k 是前面估算出来的面积）这个函数应该包含一个参数，用以表明生成多少个随机数，比如 $rsin2(10)$ 会返回一个来自这个分布的含 10 个元素的向量。用拒绝抽样法来抽样。绘制 1000 个样本的直方图。

(e) 用刚才写的函数生成 $1,000,000$ 个样本，计算均值及 95% 置信区间。(根据对称性，真实的均值应该是 π，置信区间包含了这个真实值吗？)

第 六 章

线性代数计算

线性代数用于处理向量空间中的线性运算。在数学中，通常使用列向量来表示向量，用矩阵来表示线性运算。于是对一个向量进行线性运算就变成向量与矩阵相乘，运算的操作就是矩阵相乘。

线性代数最重要的应用之一是解线性方程组。举例来说，有方程组：

$3x_1 - 4x_2 = 6$

$x_1 + 2x_2 = -3$

可以表达为：

$Ax = b$,

其中：

$A = \begin{bmatrix} 3 & -4 \\ 1 & 2 \end{bmatrix}$, $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, $b = \begin{bmatrix} 6 \\ -3 \end{bmatrix}$

上述方程组的解为：

$x = A^{-1}b = \begin{bmatrix} 0.2 & 0.4 \\ -0.1 & 0.3 \end{bmatrix} \begin{bmatrix} 6 \\ -3 \end{bmatrix} = \begin{bmatrix} 0 \\ -1.5 \end{bmatrix}$

线性代数也被用于统计中的线性回归、平滑和仿真等运算。本书会涉及到其中的部分内容，其他内容已经超出了本书的范围。

从计算的角度看，线性代数的许多问题本质上可以归结为高效准确地求解线性方程组。为了评估准确性，需要理解矩阵的性质，而这种认识本身就很有价值。效率往往意味着在严格的数学角度之外，用不同的方式来看问题。正如下面将看到的，我们不会用平常的数学方式 $x = A^{-1}b$ 来解方程组 $Ax = b$，从因为计算的角度来看，这样做既没有效率，又不准确。

在本章中，首先用几种不同的方法来解这样的问题，并用 R 来演示求解过程。在 R 中使用 LINPACK[1] 和 LAPACK[2] 两个软件包来解线性方程组。它们已经被严格测试并广受信赖，所以 R 是线性代数运算很优秀的平台。不过，在数值计算中，为了得到可信赖的结果，仍然需要理解计算过程背后的理论。

6.1 向量与矩阵

在 R 之中所说的 ``向量" 与 ``矩阵" 概念和在数学中所说的差不多。（在 R 之中，同样也允许在向量和矩阵中存储其他类型的数据，不过在本书中不会涉及。） R 通常并不区别行向量和列向量，不过在必要时， R 也确实允许区分包含一行的矩阵和包含一列的矩阵。

在 R 中数字矩阵显示为矩形的数组，但内部存储时却是存储为向量和矩阵的维度。不过进行线性代数计算时，不必在意它内部的存储形态。

6.1.1 构造矩阵对象

使用 matrix()， cbind() 和 rbind() 等函数来构造矩阵。

语法:

matrix(data, nrow, ncol) # data 是 nrow 行乘以 ncol 列的矩阵。

cbind(d1, d2, ..., dm) # d1, ..., dm 是列向量。

rbind(d1, d2, ..., dn) # d1, ..., dn 是行向量。

例 6.1

线性代数中经常会涉及到希尔伯特矩阵，原因是它们易于构造并具有一些特殊属性。

[1] Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G.W. (1978) LINPACK Users Guide. Philadelphia: SIAM.

[2] Anderson, E. et al. (1999) LAPACK Users' Guide, 3rd edition. Philadelphia: SIAM. 可从以下网址获得：www.netlib.org/lapack/lug/lapack_lug.html

```
> H3 <- matrix(c(1,1/2,1/3,1/2,1/3,1/4,1/3,1/4,1/5),nrow=3)
> H3
```

```
          [,1]      [,2]      [,3]
[1,] 1.0000000 0.5000000 0.3333333
[2,] 0.5000000 0.3333333 0.2500000
[3,] 0.3333333 0.2500000 0.2000000
```

此处 $H3$ 是一个 3×3 的矩阵，其中的元素 (i, j) 的值等于 $1/(i + j - 1)$ 。值得注意的是，命令中 ncol 参数并不是必须的，因为在 $data$ 过程中已经给向量赋了 9 个数，显然，如果如果矩阵有三行的话，就必定有三列。

同样也可以通过合并几个列向量来构建一个矩阵，例如：

```
> 1/cbind(seq(1, 3), seq(2, 4), seq(3, 5))
```

```
          [,1]      [,2]      [,3]
[1,] 1.0000000 0.5000000 0.3333333
[2,] 0.5000000 0.3333333 0.2500000
[3,] 0.3333333 0.2500000 0.2000000
```

在本例中，如果使用 $rbind()$ 函数也能得出相同的结果，因为对称性特征。

矩阵并不一定是方阵。

例 6.2

下面是一些非方阵的矩阵例子：

```
> matrix(seq(1, 12), nrow=3)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

```
> x <- seq(1, 3)
> x2 <- x^2
> X <- cbind(x, x2)
```

```
  x x2
[1,] 1  1
[2,] 2  4
[3,] 3  9
```

上面的矩阵 X 也可以使用下面的方法来构建：

```
> X <- matrix(c(1, 2, 3, 1, 4, 9), ncol=2)
```

不过它的显示稍有不同，原因是没有定义列变量名。

习题

1. 分别使用 matrix()， seq() 和 rep() 函数来构建一个 5×5 的 *Hankel* 矩阵。

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \\ 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

   将代码改写为函数，以便于能用于构建任意 $n \times n$ 维度的矩阵。用这个函数来创建 10×10 和 12×12 的 *Hankel* 矩阵。

2. 使用 rbind() 将向量 [0.1 0.5 0.0 0.4]， [0.2 0.3 0.5 0.0]， [0.3 0.0 0.5 0.2]，和 [0.2 0.3 0.2 0.3] 合并为一个 4×4 矩阵，并将结果存储在变量 P 中。

3. 使用 cbind() 函数构建一个 7×3 矩阵:

$$W = \begin{bmatrix} 1 & 2 & 4 \\ 1 & 3 & 7 \\ 1 & 4 & 5 \\ 1 & 5 & 6 \\ 1 & 6 & 7 \\ 1 & 7 & 5 \\ 1 & 8 & 3 \end{bmatrix}$$

6.1.2 读取矩阵元素，行与列名称

矩阵元素的标记与数组的标记方式一致，标号为 (i, j) 的元素就是位于第 i 行与第 j 列交叉处的元素。例如， X 矩阵中 $(3, 2)$ 的元素是 9 。如果想要读取这个元素，可以使用下列命令：

```
> X[3, 2]
```

```
[1] 9
```

可以使用 $X[i,]$ 读取矩阵 X 第 i 行的所有元素，也可以使用 $X[, j]$ 读取第 j 列的所有元素，例如：

```
> X[3,]
```

```
 x x2
 3  9
```

```
> X[, 2]
```

```
[1] 1 4 9
```

使用上述命令，得到的结果通常是向量，并不保留维度的信息。如果希望得到的结果仍然以行或列矩阵的方式来表达，需要加入选项 drop = FALSE 。

```
> X[3, , drop = FALSE]
```

```
  x x2
[1,] 3  9
```

```
> X[, 2, drop = FALSE]
```

```
  x2
[1,] 1
[2,] 4
[3,] 9
```

如矩阵 X 所显示的那样，在矩阵中行或者列都可以有变量名，例如：

```
> colnames(X)
```

```
[1] "x" "x2"
```

在这个例子中，行变量并没有名称，

```
> rownames(X)
```

```
NULL
```

此时，可以给矩阵的各行赋上名称：

```
> rownames(X) <- c("obs1", "obs2", "obs3")
> X
```

```
      x  x2
obs1  1   1
obs2  2   4
obs3  3   9
```

其实就内部来说， R 存储矩阵和数组的方式是很不同的。矩阵存储的时候是将一个向量和相关的维度一起存储，而数组只是存储为列的表单。因此，对矩阵使用 $ 符号来提取元素并不适用。例如：

```
> X$x
```

```
NULL
```

不过，可以直接使用行变量名或者列变量名来读取矩阵元素，例如：

```
> X[, "x"]
```

```
obs1 obs2 obs3
   1    2    3
```

习题

1. 构造随机矩阵[3]如下所示：

	sunny	rainy
sunny	0.2	0.8
rainy	0.3	0.7

2. 为 5 个人分别构造两个向量，分别为身高（厘米）和体重（千克）。

```
> height <- c(172, 168, 167, 175, 180)
> weight <- c(62, 64, 51, 71, 69)
```

将两个向量合并成矩阵，并将它修改成下面的形式：

	height	weight
Neil	172	62
Cindy	168	64
Pardeep	167	51
Deepak	175	71
Hao	180	69

3. 在上一题中， Pardeep 的真实身高是 162 厘米， Hao 的真实身高是 181 厘米，体重是 68 千克。请修正上面矩阵的错误。

[3]随机矩阵的特征是，所有的元素的值都非负，各行元素的和为 1 。这种矩阵用于表示马尔科夫链中的转移概率。在本例中，需要表达的是不同日子间天气状况的转移概率，如果今天是晴天，那么明天是晴天的概率是 0.2 ，如果今天下雨，那么明天是晴天的概率是 0.3 。

6.1.3 矩阵属性

矩阵的维度是矩阵行和列的数量，例如：

```
> dim(X)
```

```
[1] 3 2
```

对于一个 2×2 的行列式 $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$，可以使用 $ad - bc$ 来计算。对于更大的方阵，计算起了比较复杂。不过在 R 中可以使用 det() 函数来计算。

```
> det(H3)
```

```
[1] 0.000462963
```

要获取矩阵对角线上的元素，可以使用 diag() 函数，例如:

```
> diag(X)
```

```
[1] 1 4
```

```
> diag(H3)
```

```
[1] 1.0000000 0.3333333 0.2000000
```

可以自己构造一个函数来计算矩阵的迹：

```
> trace <- function(data) sum(diag(data))
```

把这个函数应用于在例 6.1 和例 6.2 中，可得：

```
> trace(X)
```

```
[1] 5
```

```
> trace(H3)
```

```
[1] 1.533333
```

使用 diag() 函数，还可以将一个向量转换成对角矩阵，对角线上的元素恰好就是向量中的元素。例如：

```
> diag(diag(H3))
```

```
     [,1]      [,2] [,3]
[1,]    1 0.0000000  0.0
[2,]    0 0.3333333  0.0
[3,]    0 0.0000000  0.2
```

使用 t() 函数，可以计算矩阵的转置 X^T ：

```
> t(X)
```

```
     obs1 obs2 obs3
x       1    2    3
x2      1    4    9
```

习题

1. 证明 $det(A) = det(A^T)$ ，并用多个 A 来验证。
2. 如果矩阵 A 满足 $A^T = -A$ ，那么这个矩阵称为斜对称 (skew-symmetric) 矩阵。请构造一个 3×3 的斜对称矩阵，并证明它行列式的值是 0 。该矩阵的迹是多少？

6.1.4 三角矩阵

使用 lower.tri() 函数和 uper.tri() 可以构造下三角矩阵和上三角矩阵。函数输出的矩阵包含逻辑元素，其中 TRUE 代表对应的三角矩阵元素。例如：

```
> lower.tri(H3)
```

```
     [,1]  [,2]  [,3]
[1,] FALSE FALSE FALSE
[2,]  TRUE FALSE FALSE
[3,]  TRUE  TRUE FALSE
```

可以用下面的方法得到 $H3$ 矩阵所对应的下三角矩阵，其余元素都置为 0 。

```
> Hnew <- H3
> Hnew[upper.tri(H3, diag=TRUE)] <- 0 # diag=TRUE 包含对角线上元素
> Hnew
```

```
          [,1]   [,2] [,3]
[1,] 0.0000000 0.00    0
[2,] 0.5000000 0.00    0
[3,] 0.3333333 0.25    0
```

习题

1. 生成一个矩阵，使其的上三角矩阵元素和 $H3$ 的元素（包括对角元素）一致，对角线之下的元素都为 0 。
2. 查看下述代码的输出，

```
> Hnew[lower.tri(H3)]
```

   与您期望的结果一致吗？

3. 使用 6.2 节中定义的 X ，那么 $X[3,2]$ 和 $X[3,2,drop = FALSE]$ 两条命令的结果有何不同？使用 R 计算两个结果的维度。

6.1.5 矩阵算术

矩阵乘以一个常数的操作和向量乘以一个常数的操作一样。例如对于矩阵 X ，可以用下面的方法将每个元素都乘以 2 ：

```
> Y <- 2 * X
> Y
```

```
     x  x2
obs1 2   2
obs2 4   8
obs3 6  18
```

矩阵逐个元素相加的计算方法和矩阵相加的方法也是一样的，例如：

```
> Y + X
```

```
     x  x2
obs1 3   3
obs2 6  12
obs3 9  27
```

当进行矩阵相加时，要确保两个矩阵的维度相符。如果不相符的话，就会返回出错信息：

```
> t(Y) + X
```

```
Error in t(Y) + X : non-conformable arrays
```

在本例中， Y 的转置是 2×3 ，而 X 的维度是 3×2 。

如果输入指令 $X * Y$ ，那么执行的是元素间的相乘。请注意，它和下面将要谈到的矩阵相乘是不同的。例如：

```
> X * Y
```

```
      x   x2
obs1  2    2
obs2  8   32
obs3 18  162
```

需要再次强调的是，为了进行这样的矩阵相乘，矩阵的维度必须相符合。

6.2 矩阵乘法与逆矩阵

如果 A 和 B 是两个矩阵， AB 代表矩阵的乘积，它是两个运算的综合，首先应用于 B ，然后将结果应用于 A 。矩阵相乘时，两个矩阵的维度必须匹配，即第一个矩阵的列数必须等于第二个矩阵的行数。最后生成的矩阵 AB 的维度具有矩阵 A 的行数和矩阵 B 的列数。

在 R 中，矩阵相乘的运算符是 $\%*\%$ ，例如:

```
> t(Y) %*% X
```

```
   x  x2
x  28  72
x2 72 196
```

在前文中 $t(Y)$ 有三列，而 X 有三行，所以他们可以相乘，其结果是 2×2 的矩阵，因为 $t(Y)$ 有两行，X 有两列。如果不对 Y 进行转置，那么它们相乘就返回出错信息：

```
> Y %*% X
```

```
Error in Y %*% X : non-conformable arguments
```

使用 crossprod() 可以更有效地计算 $Y^T X$ 。

```
> crossprod(Y,X)
```

```
   x  x2
x  28  72
x2 72 196
```

请注意在函数 crossprod() 中的第一个参数会自动被转置。这个函数效率之所以更高，是因为在计算 $t(Y)\%*\%X$ 时，首先要创建一个新对象 $t(Y)$ 。如果 Y 是一个很大的矩阵，那么上述过程会消耗较多的内存和较多的计算时间。但 crossprod() 函数直接调用 Y ，因为在 Y^T 中的元素 (i, j) ，就是 Y 中的元素 (j, i) 。

习题

1. 使用 6.2 节中的矩阵 X，计算 $1.5X$。

2. 使用 crossprod() 函数计算 X^TX 和 XX^T。计算结果的维度分别是什么？

6.2.1 逆矩阵

设 A 是 $n \times n$ 的矩阵，它的逆矩阵记作 A^{-1}。逆矩阵的获取是通过求解 $AA^{-1} = I$ 得到的，其中 I 是单位矩阵。我们可以将它们看做包含 n 个未知数的 n 个独立的线性方程组，求解这个方程组就能得到逆矩阵 A^{-1}。例如：

$A = \begin{bmatrix} 3 & -4 \\ 1 & 2 \end{bmatrix}$

构建矩阵等式：

$\begin{bmatrix} 3 & -4 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

它等价于这样两个方程：

$\begin{bmatrix} 3 & -4 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$，和

$\begin{bmatrix} 3 & -4 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} b_{12} \\ b_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$，

通常采用求解上面两个方程的方法来得到 A^{-1}。

不过这种方法并不总是最好的，一般来说，求 A^{-1} 的目的是为了得到方程组 $Ax = b$ 的解：$x = A^{-1}b$。从计算的角度来看，解 n 个线性方程组只是为了得到一个结果并将它用于解一个线性方程组，这种做法是讲不通的。如果知道如何解方程组，那么就该用这些方法直接去解 $Ax = b$。更进一步来说，采用 A^{-1} 比使用直接求解的方法得到的结果可能更差，因为这个过程需要太多的操作，所以有更多的机会将舍入误差带到最后的结果中。

6.2.2 LU 分解

求解方程组 $Ax = b$ 的基本策略是将问题简化，这往往涉及到将矩阵 A 重新改写成一种特殊形式，其中的一种称为 LU 分解。

在 LU 分解时，将矩阵 A 改写成两个矩阵 L 和 U 的乘积。其中 L 是下三角矩阵，它的对角线都是 1，如：

$$L = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \ddots & \vdots \\ \vdots & & \ddots & 0 \\ l_{n1} & l_{n2} & \cdots & 1 \end{bmatrix}$$

U 是上三角矩阵：

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{nn} \end{bmatrix}$$

事实证明，只要按照一步一步的操作，就很容易将 A 分解，请看下面的例子：

$$A = \begin{bmatrix} 2 & 4 & 3 \\ 6 & 16 & 10 \\ 4 & 12 & 9 \end{bmatrix},$$

计算过程如下所示，我们将矩阵 A 通过元素 a_{ij} 的形式来表示，重复执行这样的运算：

$$a_{ij} = \sum_{k=1}^{3} l_{ik}u_{kj}$$

然后利用所知道的在 L 和 U 中的 0 和 1 可以计算：

1. $a_{11} = 2 = l_{11} \times u_{11} = 1 \times u_{11}$, 所以 $u_{11} = 2$
2. $a_{21} = 6 = l_{21} \times u_{11} = l_{21} \times 2$, 所以 $l_{21} = 3$
3. $a_{31} = 4 = l_{31} \times u_{11} = l_{31} \times 2$, 所以 $l_{31} = 2$

4. $a_{12} = 4 = l_{11} \times u_{12}$, 所以 $u_{12} = 4$
5. $a_{22} = 16 = l_{21} \times u_{12} + l_{22} \times u_{22} = 3 \times 4 + 1 \times u_{22}$, 所以 $u_{22} = 4$
6. $a_{32} = 12 = l_{31} \times u_{12} + l_{32} \times u_{22} = 2 \times 4 + l_{32} \times 4$, 所以 $l_{32} = 1$
7. $a_{13} = 3 = l_{11} \times u_{13} = 1 \times u_{13}$, 所以 $u_{13} = 3$
8. $a_{23} = 10 = l_{21} \times u_{13} + l_{22} \times u_{23} = 3 \times 3 + 1 \times u_{23}$, 所以 $u_{23} = 1$
9. $a_{33} = 9 = l_{31} \times u_{13} + l_{32} \times u_{23} + l_{33} \times u_{33} = 2 \times 3 + 1 \times 1 + 1 \times u_{33}$, 所以 $u_{33} = 2$

一旦得到了 L 和 U，那么解方程组 $Ax = b$ 就很容易了。将方程组写成 $L(Ux) = b$，并使 $y = Ux$，从方程 $Ly = b$ 中解出 y。因为 L 是下三角矩阵，所以这是向前消元法的简单应用。继续上面的例子，因为 $b = [-1, -2, -7]^T$，并设 $y = [y_1, y_2, y_3]^T$，用下面的关系式 $b_i = \sum_{j=1}^{3} l_{ij} y_j$ 来计算：

10. $b_1 = -1 = l_{11} \times y_1 = 1 \times y_1$, 所以 $y_1 = -1$
11. $b_2 = -2 = l_{21} \times y_1 + l_{22} \times y_2 = 3 \times (-1) + 1 \times y_2$, 所以 $y_2 = 1$
12. $b_3 = -7 = l_{31} \times y_1 + l_{32} \times y_2 + l_{33} \times y_3 = 2 \times (-1) + 1 \times 1 + 1 \times y_3$, 所以 $y_3 = -6$

最后来求 $Ux = y$。这里 U 是上三角矩阵，使用向后消元法会显得更容易一点。

13. $y_3 = -6 = u_{33} \times x_3 = 2 \times x_3$， 所以 $x_3 = -3$
14. $y_2 = 1 = u_{22} \times x_2 + u_{23} \times x_3 = 4 \times x_2 + 1 \times (-3)$， 所以 $x_2 = 1$
15. $y_1 = -1 = u_{11} \times x_1 + u_{12} \times x_2 + u_{13} \times x_3 = 2 \times x_1 + 4 \times 1 + 3 \times (-3)$， 所以 $x_1 = 2$

通过持续使用上述步骤后，求解方程组 $Ax = b$ 就被分解成求 15 个相应的线性方程，每个方程中只有一个未知数。这个过程很容易自动执行。事实上，在 R 中求解线性方程组的默认方式就是基于这样的算法，唯一的不同是在求解之前，列的顺序被重新排列以确保舍入误差最小化。

6.2.3 求逆矩阵

在 R 之中，使用 solve() 函数或者 qr.solve() 函数来求逆矩阵或解线性方程组。其中 solve() 函数使用 LU 分解算法， qr.solve() 使用 QR 分解的算法。

下面用 6.1 节中 3×3 的希尔伯特矩阵为例，来求其逆矩阵。

```
> H3inv <- solve(H3)
> H3inv
```

```
     [,1] [,2] [,3]
[1,]    9  -36   30
[2,]  -36  192 -180
[3,]   30 -180  180
```

要验证上面的结果，可以将两个矩阵相乘，其结果应该是一个 3×3 的单位矩阵。

```
> H3inv %*% H3
```

```
              [,1]          [,2]          [,3]
[1,] 1.000000e+00  8.881784e-16  6.882515e-16
[2,] -3.774758e-15 1.000000e+00 -3.420875e-15
[3,] 6.144391e-15  0.000000e+00  1.000000e+00
```

结果显示，对角线上的元素都是 1，但是非对角线上的五个元素并不是零，它们使用了科学计数法来表示，而且都小于 10^{-14}。从数值上来看，它们都接近于 0。所以 $H3inv$ 并不是 $H3$ 的逆矩阵，不过我们相信它非常接近于真实的逆矩阵。

习题

1. 求 $X^T X$ 的逆矩阵，并用 crossprod() 函数来验证。
2. 可以计算 XX^T 的逆矩阵么？为什么？

3. 一个 $n \times n$ 的希尔伯特矩阵，它的元素 (i, j) 的值等于 $1/(i+j-1)$ 。

   (a) 编写一个函数用以生成 $n \times n$ 的希尔伯特矩阵，其中 n 是任意正整数。

   (b) 是否所有的希尔伯特矩阵都可逆？

   (c) 使用 solve() 和 qr.solve() 函数顺次计算 $n = 10$ 之前希尔伯特矩阵的逆矩阵，有没有发现什么问题？

6.2.4 求线性方程组

函数 solve(A,b) 可用来求解线性方程组 $Ax = b$ 。例如，可以求 $H_3 x = b$ 中 x 的解。其中 H_3 是一个 3×3 的希尔伯特矩阵， $b = [1, 2, 3]^T$ 。

```
> b <- c(1, 2, 3)
> x <- solve(H3, b)
> x
```

```
[1] 27 -192 210
```

换句话说， x 的解是向量 $x = [27, -192, 210]^T$ 。

习题

1. 设 $[x_1, x_2, x_3, x_4, x_5, x_6]^T = [10, 11, 12, 13, 14, 15]^T$ 。求五次方程 $f(x)$ 的系数，使得 $[f(x_1), f(x_2), f(x_3), f(x_4), f(x_5), f(x_6)]^T = [25, 16, 26, 19, 21, 20]^T$ 。(提示：五次函数 $f(x) = a_0 + a_1 x + a_2 x_2 + a_3 x_3 + a_4 x_4 + a_5 x_5$ 可以看做是一个行向量 $[1, x, x_2, x_3, x_4, x_5]$ 与一个列向量 $[a_0, a_1, a_2, a_3, a_4, a_5]^T$ 相乘的结果。给定 $[f(x_1), f(x_2), f(x_3), f(x_4), f(x_5), f(x_6)]^T$ 之后求解这个矩阵即可。)

6.3 特征值与特征向量

求矩阵的特征值与特征向量，可以使用 eigen() 函数。例如：

```
> eigen(H3)
```

```
$values
[1] 1.408318927 0.122327066 0.002687340
$vectors
           [,1]       [,2]       [,3]
[1,] 0.8270449  0.5474484  0.1276593
[2,] 0.4598639 -0.5282902 -0.7137469
[3,] 0.3232984 -0.6490067  0.6886715
```

解读一下这个输出结果，设 x_1 表示 $\$vectors$ 中的第一列，即 $[0.8270.4590.323]^T$ ，它表示第一个特征向量，与它相对应的特征值是 1.408 。这样

$H_3 x_1 = 1.408 x_1$。

用 x_2 与 x_3 分别表示 $\$vectors$ 中的第二列和第三列，那么可得：

$H_3 x_2 = 0.122 x_2$，

以及

$H_3 x_3 = 0.00268 x_3$。

习题

1. 计算矩阵 $H = X(X^TX)^{-1}X^T$，其中 X 参见第 6.1 节中定义。
2. 计算矩阵 H 的特征向量与特征值。
3. 计算矩阵 H 的迹，并将它与特征值的和相比较。
4. 计算矩阵 H 行列式的值，并与特征值的乘积相比较。
5. 根据定义，证明 X 与 $I - H$ 的列是 H 的特征向量。
6. 生成一个 6×6 的希尔伯特矩阵，计算其特征值和特征向量。计算其逆矩阵。原矩阵与逆矩阵的特征值有什么关系吗？应该有什么关系吗？请用 7×7 的希尔伯特矩阵重复上面的问题。

6.4 高级主题

6.4.1 矩阵的奇异值分解

一个方阵 A 的奇异值分解，可分解成三个方阵, U , D 和 V 。其中 D 是对角矩阵，他们之间的关系是：$A = UDV^T$

矩阵 U 和 V 是*正交*的，因为 $U^{-1} = U^T$ ； $V^{-1} = V^T$ 。

奇异值分解技术通常用于求线性方程组的精确解。矩阵 D 中的元素被称为 A 的奇异值。由于 $A^TA = V^{-1}D^2V$ ，这是一个“相似变换”，它意味着， A 奇异值的平方是 A^TA 的特征值。

可以使用 svd() 函数进行奇异值分解。例如，对一个 3×3 的希尔伯特矩阵 H_3 进行奇异值分解是：

```
> H3.svd <- svd(H3)
> H3.svd
```

```
$d
[1]    1.408318927    0.122327066    0.002687340
$u
              [,1]          [,2]          [,3]
[1,]    -0.8270449     0.5474484     0.1276593
[2,]    -0.4598639    -0.5282902    -0.7137469
[3,]    -0.3232984    -0.6490067     0.6886715
$v
              [,1]          [,2]          [,3]
[1,]    -0.8270449     0.5474484     0.1276593
[2,]    -0.4598639    -0.5282902    -0.7137469
[3,]    -0.3232984    -0.6490067     0.6886715
```

通过对上面的结果按一定方式相乘可以重构 H_3 ，以此来对函数进行验证。

```
> H3.svd$u %*% diag(H3.svd$d) %*% t(H3.svd$v)
```

```
             [,1]         [,2]         [,3]
[1,]    1.0000000    0.5000000    0.3333333
[2,]    0.5000000    0.3333333    0.2500000
[3,]    0.3333333    0.2500000    0.2000000
```

由于矩阵 U , V 和 D 的特性，奇异值分解提供了一个计算逆矩阵的简单方法。例如， $H_3^{-1} = VD^{-1}U^T$ ，就可以用下面的方法来计算。

```
> H3.svd$v %*% diag(1/H3.svd$d) %*% t(H3.svd$u)
```

```
     [,1] [,2] [,3]
[1,]    9  -36   30
[2,]  -36  192 -180
[3,]   30 -180  180
```

6.4.2 正定矩阵的乔莱斯基分解

如果一个矩阵 A 是正定的，那么它拥有一个平方根。事实上，通常有几个矩阵 B，使得 $B^2 = A$。乔莱斯基分解 (Choleski) 与此相似，不过它是设法找到一个上三角矩阵 U，使得 $U^TU = A$。函数 chol() 执行这个功能。

例如对一个 3×3 的希尔伯特矩阵进行乔莱斯基分解。

```
> H3.chol <- chol(H3)
> H3.chol    # 这是上三角矩阵 U
```

```
     [,1]      [,2]      [,3]
[1,]    1 0.5000000 0.3333333
[2,]    0 0.2886751 0.2886751
[3,]    0 0.0000000 0.0745356
```

```
> crossprod(H3.chol, H3.chol) # U^T U 重新得到 H3
```

```
     [,1]      [,2]      [,3]
[1,]    1 0.5000000 0.3333333
[2,]  0.5 0.3333333 0.2500000
[3,]  0.3 0.2500000 0.2000000
```

一旦通过 $A = U^TU$ 得到乔莱斯基分解，就可以通过 $A^{-1} = U^{-1}U^{-T}$ 来计算逆矩阵（其中 U^{-T} 表示 U^{-1} 的转置）。这种计算方法比高斯消元法要稳定得多。我们可以使用 chol2inv() 来执行上述计算，例如计算希尔伯特矩阵 H_3 的逆矩阵：

```
> chol2inv(H3.chol)
```

```
     [,1] [,2] [,3]
[1,]    9  -36   30
[2,]  -36  192 -180
[3,]   30 -180  180
```

通过乔莱斯基分解，可以计算类似 $Ax = b$ 的线性方程。如果 $A = U^TU$，那么可得 $Ux = U^{-T}b$，x 的值可以通过下面两个步骤得到：

1. 求解 $U^Ty = b$，它的解满足 $y = U^{-T}b$。
2. 求解 $Ux = y$。

第一个方程组是下三角矩阵，所以可以使用 forwardsolve() 函数的向前消元法来求解。第二个是上三角矩阵，所以使用 backsolve() 函数的向后替代法求解。

对于 $H_3x = b$， $b = [123]^T$ 的问题，可以用下述方法求解：

```
> b <- seq(1, 3)
> y <- forwardsolve(t(H3.chol), b)
> backsolve(H3.chol, y) # 得到 x 的解
```

```
[1]   27 -192  210
```

6.4.3 矩阵的 QR 分解

另一个分解矩阵 A 的方法是 QR 分解， $A = QR$ 。其中 Q 是一个正交矩阵，R 是上三角矩阵。即使 A 不是方阵的时候，也可以使用这种分解方法。需要强调的是，这种分解方法能解出线性方程组的精确解。

例如，假如要解下面的方程中的 x

$Ax = b$

其中，A 是 $n \times n$ 的矩阵，b 是 n 维的向量。如果首先对 A 进行 QR 分解，那么可以将上述方程写成

$QRx = b.$

两边都乘上 Q^T 可得

$Rx = Q^Tb$

因为 R 是个上三角矩阵，所以这个方程组更容易求解。而且还注意到，Q^Tb 是很容易计算的 n 维向量。

在计算中，需要用到 qr() 函数。例如：

```
> H3.qr <- qr(H3)
> H3.qr
```

```
$qr
           [,1]       [,2]        [,3]
[1,] -1.1666667 -0.6428571 -0.45000000
[2,]  0.4285714 -0.1017143 -0.105337032
[3,]  0.2857143  0.7292564  0.003901372

$rank
[1] 3

$qraux
[1] 1.857142857 1.684240553 0.003901372

$pivot

[1] 1 2 3


attr(,"class")

[1] "qr"
```

返回的结果是属于 qr 类的对象。

进一步将函数 $qr.Q()$ 和 $qr.R()$ 用于这个对象，就能得到精确的 Q 和 R, 例如：

```
> Q <- qr.Q(H3.qr)
> Q
```

```
           [,1]       [,2]       [,3]
[1,] -0.8571429  0.5016049  0.1170411
[2,] -0.4285714 -0.5684856 -0.7022469
[3,] -0.2857143 -0.6520864  0.7022469
```

```
> R <- qr.R(H3.qr)
> R
```

```
          [,1]       [,2]         [,3]
[1,] -1.166667 -0.6428571 -0.450000000
[2,]  0.000000 -0.1017143 -0.105337032
[3,]  0.000000  0.0000000  0.003901372
```

将 Q 和 R 相乘，可以重新得到 $H3$:

```
> Q %*% R
```

```
          [,1]      [,2]      [,3]
[1,] 1.0000000 0.5000000 0.3333333
[2,] 0.5000000 0.3333333 0.2500000
[3,] 0.3333333 0.2500000 0.2000000
```

同时，可以用 $R^{-1}Q^T$ 来得到 $H3$ 的逆矩阵。由于 R 是一个上三角矩阵，所以这种计算方法理论上更快。下面我们用一种效率不是很高的方法来计算 R^{-1}，只是为了说明用分解法能计算逆矩阵。

```
> qr.solve(R) %*% t(Q)
```

```
     [,1] [,2] [,3]
[1,]    9  -36   30
[2,]  -36  192 -180
[3,]   30 -180  180
```

6.4.4 矩阵的条件值

函数 kappa() 用于计算给定矩阵的条件值（最大和最小非零奇异值的比值）。通过这个值可以评估对矩阵进行数值运算时，结果到底有多糟糕。一个大的条件值意味着很差的数值属性。

```
> kappa(H3)
```

```
[1] 646.2247
```

这个值很大，说明计算逆矩阵得到的结果会很不精确。

6.4.5 外积

函数 outer() 有时在统计计算中非常有用。它可用于计算来自两个向量的任何一对元素。一个简单的例子包括计算从 1 到 5 的数之间任何一对组合的商。

```
> x1 <- seq(1, 5)
> outer(x1, x1, "/") # 或者 outer(x1, x1, function(x, y) x / y)
```

```
     [,1] [,2]      [,3] [,4] [,5]
[1,]    1  0.5 0.3333333 0.25  0.2
[2,]    2  1.0 0.6666667 0.50  0.4
[3,]    3  1.5 1.0000000 0.75  0.6
[4,]    4  2.0 1.3333333 1.00  0.8
[5,]    5  2.5 1.6666667 1.25  1.0
```

如果将除号改成减号，则可以得到任意一对组合之间的差。

```
> outer(x1, x1, "-")
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    0   -1   -2   -3   -4
[2,]    1    0   -1   -2   -3
[3,]    2    1    0   -1   -2
[4,]    3    2    1    0   -1
[5,]    4    3    2    1    0
```

第三个参数可以是任意一个能用于两个向量的函数。而第二个参数也可以不同于第一个参数。例如：

```
> y <- seq(5, 10)
> outer(x1, y, "+")
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    6    7    8    9   10   11
[2,]    7    8    9   10   11   12
[3,]    8    9   10   11   12   13
[4,]    9   10   11   12   13   14
[5,]   10   11   12   13   14   15
```

6.4.6 克罗内克积

函数 kronecker() 用于计算两个矩阵之间的克罗内克积。详情可以用 help() 查看帮助文件。

6.4.7 APPLY() 函数

有时候，要用同一个函数来计算矩阵中的每一行或每一列，虽然可以用 for() 函数来作一个循环计算，但效率更高的方法是用 apply() 函数。

这个函数包含三个参数，第一个参数是指定要计算的矩阵。第二个参数是指出是对行还是对列进行操作，分别用 1 和 2 来指代。第三个参数标明选用的函数。

下面的例子是对 H_3 矩阵的每一行求和。

```
> apply(H3, 1, sum)
```

```
[1] 1.8333333 1.0833333 0.7833333
```

本章习题

1. 假设有如下循环行列式：

$$P = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 0.4 & 0.1 & 0.2 & 0.3 \\ 0.3 & 0.4 & 0.1 & 0.2 \\ 0.2 & 0.3 & 0.4 & 0.1 \end{bmatrix}$$

   (a) P 是随机矩阵之一，试用 apply() 函数来证明这个矩阵每一行的和都等于 1。

   (b) 计算 P^n，其中 $n = 2, 3, 5, 10$。能从中找到什么规律吗？

   (c) 找出一个非负的向量 x，使得各元素之和等于 1，并且能满足 $(I - P^T)x = 0$。能从中找到 P^{10} 和 x 之间的联系吗？

   (d) 构建一个循环，从集合 $\{1, 2, 3, 4\}$ 中生成伪随机序列 y，使它满足下面的规则：

      (i) 使得 $y_1 \leftarrow 1$。

      (ii) 当 $j = 2, 3, \ldots, n$ 时，使得 $y_j = k$ 的概率等于 $P_{y_{j-1},k}$。例如，y_2 有 0.1 的可能性是 1，有 0.2 的可能性是 2，等等。把 n 设为一个比较大的数，比如 10000。那么向量 y 就是一个仿真的马尔科夫链。

   (e) 用 table() 函数列出向量 y 四个可能值的相对频率分布。把这个分布和先前得到的*静态分布* 的 x 作比较。

2. 使用一个新的 P 矩阵，然后重复上面的所有计算。

$$P = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 & 0.0 & 0.0 & 0.0 \\ 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.4 \\ 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.0 & 0.0 \\ 0.3 & 0.3 & 0.3 & 0.1 & 0.0 & 0.0 & 0.0 \\ 0.3 & 0.3 & 0.3 & 0.1 & 0.0 & 0.0 & 0.0 \\ 0.3 & 0.3 & 0.3 & 0.1 & 0.0 & 0.0 & 0.0 \\ 0.3 & 0.3 & 0.3 & 0.1 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

3. 一个保险公司有四大类保险策略，我们分别用 A, B, C, D 表示

   - 他们总共有 245921 种保险策略。
   - A 类策略中每年收益 10 美元，B 类策略每年收益 30 美元，C 类策略中每年收益 50 美元，D 类策略每年收益 100 美元。
   - 所有策略的年总收入是 7304620 美元。
   - 每年申请索赔的概率按不同的速度上升。A 类策略每年的索赔率上升 0.1，B 类上升 0.15，C 类上升 0.03，D 类上升 0.5。

- 每年总的预期索赔量是 34390.48。
- 不同类策略索赔金额不同，A 类是 50 美元，B 类是 180 美元，C 类是 1500 美元，D 类是 250 美元。
- 总的索赔金额是 6864693，它是每种策略期望的索赔金额乘以索赔数量的加总。

使用 R 来计算下面的问题。

(a) 求出每一类策略的数量。

(b) 求出每一类的总收入和预期索赔金额。

(c) 假设索赔的上升符合泊松过程，每一笔索赔金额符合 Gamma 分布（均值已经在上面列出，外形参数是 2）。用仿真方法来计算下面的问题：

(i) 总索赔额的方差。

(ii) 总索赔额超过总收入的概率。

写一个函数来做上面的计算，首先计算公司的总收入和总索赔额，然后计算每一类策略的总收入和总索赔额。

第 七 章

数值优化

在统计学和应用数学的很多领域，面临着类似的问题：给定函数 $f(\cdot)$，当自变量 x 变化时，怎样使 $f(x)$ 的值最大或最小。

例如在金融模型中，x 可以是股票、债券等投资产品的数量，$f(x)$ 是指一个投资组合的预期收益。x 也可以有某些约束条件（比如投资量必须为正，总投资额为不能超过一定额度等）。

在统计模型中，希望找到模型的参数，最大限度地减少了模型的预测误差。这里 x 是模型的参数，$f(\cdot)$ 是预测误差的测算方式。

在实际中，知道如何最小化就足够了。因为如果想要最大化 $f(x)$ 的话，只需要改变它的符号，并将其最小化即可。我们将最大化和最小化都称作 ``数值优化"。通过求导数或者其他一些数学运算，常常可以解这样的优化问题，不过有时候也有困难，因为函数 $f(\cdot)$ 和参数 x 形式多种多样，致使计算过程很繁琐。

7.1 黄金分割搜索法

假设一个单变量函数在区间 $[a,b]$ 内有唯一最小值，对这类问题，黄金分割搜索法就是找到最小值的简单方法。

若需要求下面函数的最小值：

$f(x) = |x - 3.5| + (x - 2)^2$

已知自变量区间为 $[0,5]$ 。由于函数在 $x = 3.5$ 时不可导，所以在求最小值时就要特别注意。我们可以构建一个 R 函数来求解上面这个问题：

```
f <- function(x) {abs(x-3.5)+(x-2)^2}
```

可以使用 curve() 命令绘制函数的图形，这样可以很直观地查看函数在区间内是否有单一最小值。

```
curve(f, from = 1, to = 5)
```

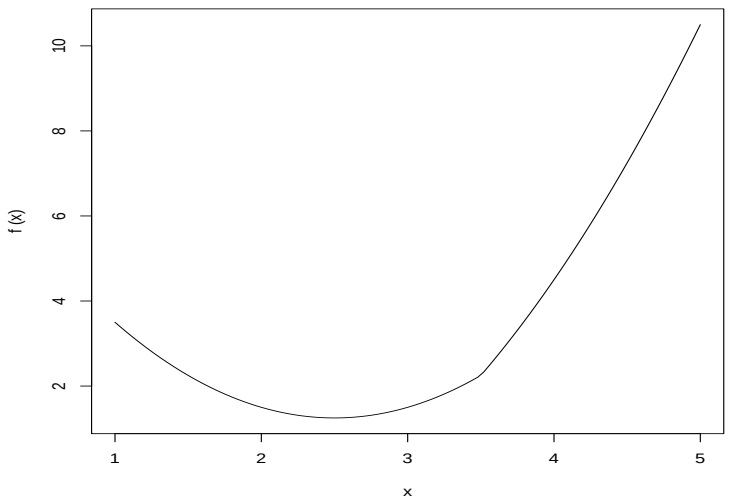


图 7.1: $f(x) = |x - 3.5| + (x - 2)^2$

图7.1 显示了函数的曲线，从图中可以看到最小值大约出现在 $x = 2.5$ 处。
黄金分割搜索法是一个迭代方法，该方法可概述如下：

1. 从已知含有极小值的区间 $[a, b]$ 开始。
2. 找到一个更小的区间 $[a', b']$ 使得该区间仍包含极小值，并重复这一过程。
3. 当 $b' - a'$ 的值小于预先设定值时，停止迭代。

当搜索停止时，最后一次迭代区间的中点可看做最小值很好的估计，这个估计值的最大误差为 $(b'-a')/2$ 。
在第二步计算开始前，先在区间 $[a, b]$ 内选两点 $x_1 < x_2$ （至于如何选这两点，会在下面详述），分别计算函数在这两点处的值。因为假定函数在整个区间内只有一个最小值，所以如果 $f(x_1) > f(x_2)$ ，那么函数的最小值一定位于 x_1 的右边，即新得到的区间 $[a', b'] = [x_1, b]$ 。同样如果 $f(x_1) < f(x_2)$ ，那么包含最小值的新区间为 $[a', b'] = [a, x_2]$ （见图7.2）。（假如两个函数的值恰好相等怎么办？我们会在下面进一步讨论）。此时得到一组新的数字， x_1 ， $f(x_1)$ ， x_2 和 $f(x_2)$ 。重复上面的过程，直到满足先前设定好的标准。

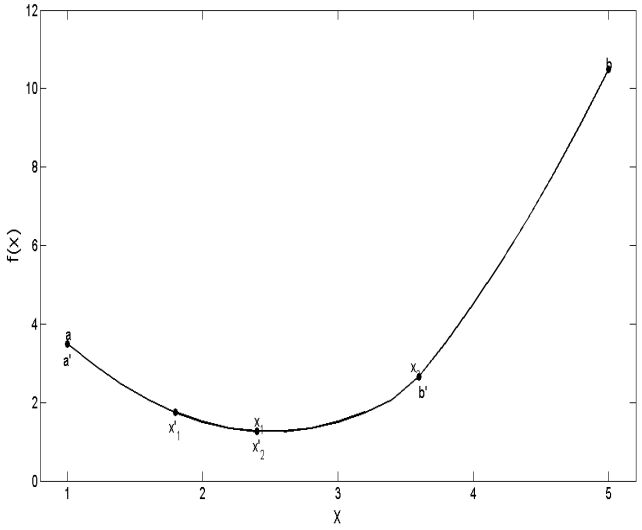


图 7.2: 对 $f(x) = |x - 3.5| + (x - 2)^2$ 使用黄金分割法一次迭代

在 a 和 b 之内确定新的内插点，依据的是黄金分割比例 $\phi = (\sqrt{5}+1)/2$ 。黄金分割率（在第三章关于斐波那契数时曾涉及）有很多有趣的数学性质，比如后面要用的这样的属性： $1/\phi = \phi - 1$ ， $1/\phi^2 = 1 - 1/\phi$ （有些人将 $\Phi = 1/\phi$ 称作 ``白银比率"，不过我们将继续使用 ϕ ）。
用下面的方法来选择新的内插点 $x_1 = b - (b-a)/\phi$ ， $x_2 = a + (b-a)/\phi$ 。理由是，设定了一个内插点之后，有可能用它来替代 a ，即 $a' = x_1$ 。所以 x_1 的值就等于：

$$\begin{aligned} x_1' &= b - (b - a')/\phi \\ &= b - (b - x_1)/\phi \\ &= b - (b - a)/\phi^2 \\ &= a + (b - a)/\phi \\ &= x_2 \end{aligned} \tag{7.1}$$

这意味着可以重复利用一些已经存在的点，而不必重新计算。比如不必计算 $f(x_1')$ 而可以直接用 $f(x_2)$ 的值。相应的，如果把 x_2 更新到 b' ， $b' = x_2$ ，那么 $x_2' = x_1$ 。
把这些特征整合到下面的 R 函数中：

```
> golden <- function (f, a, b, tol = 0.0000001)
+ {
+     ratio <- 2/(sqrt(5)+1)
```

```
+    x1 <- b-ratio*(b-a)
+    x2 <- a+ratio*(b-a)
+    f1 <- f(x1)
+    f2 <- f(x2)
+    while(abs(b-a)>tol)  {
+         if (f2>f1) {
+         b <- x2
+         x2 <- x1
+         f2 <- f1
+         x1 <- b-ratio*(b-a)
+         f1 <- f(x1)
+    } else {
+         a <-  x1
+         x1 <- x2
+         f1 <- f2
+         x2 <- a+ratio *(b-a)
+         f2 <- f(x2)
+         }
+    }
+    return((a+b)/2)
+ }
```

经过测试，发现 golden() 函数至少对 f 是有效的：

```
golden(f, 1, 5)
```

```
[1] 2.5
```

习题

1. 将黄金分割搜索法应用于下列函数，求最小值。

   (a) $f(x) = |x - 3.5| + |x - 2| + |x - 1|$

   (b) $f(x) = |x - 3.2| + |x - 3.5| + |x - 2| + |x - 1|$

   对于第二个函数，通过图形可以看出最小值有多个。请证明使用 golden() 函数求最小值时，依赖于设定的函数区间。

2. 对于奇数个数字 $x_1, x_2, \ldots, x_n$，下面函数的最小值就是中间那个数字的数值（习题 1(a) 就是这样的例子）。

$$f(x) = \sum_{i=1}^{n} |x - x_i|$$

   用下面的数字来验证这个命题。

   (a) 3, 7, 9, 12, 15

   (b) 3, 7, 9, 12, 15, 18, 21

   假如数字是偶数个的话，结果又会是如何呢？

3. 利用黄金搜索法构建一个函数并用来求函数的极大值。

7.2 牛顿 -拉夫逊法

如果目标函数有连续的一阶和二阶导数，并且知道如何求这些导数，那么就可以用这个性质构建一种比黄金分割搜索法更快的算法，来求目标函数的最小值。

假设要在区间 $[a,b]$ 之间寻找能使函数 $f(x)$ 最小化的 x^*，如果函数的最小值不是出现在 a 点或者 b 点，那么在 x^* 点肯定能满足 $f'(x^*)=0$。这是求最极值的必要条件，但这还不够，我们还要验证 x^* 是不是真的最小化函数 $f(x)$。因为能满足 $f'(x^*)=0$ 的点也可能是极大值或者是拐点。所以还需要检验它的充分条件，即 $f''(x^*)>0$。

假设我们猜测在 x_0 时函数得到最小值，那么可以在这一点展开泰勒级数：

$f'(x)\approx f'(x_0)+(x-x_0)f''(x_0)$

令等式右侧等于零可以求得 $f'(x^*)=0$ 的近似值。

用这种方法可以构建牛顿 -拉夫逊算法。先设定一个初始估计值 x_0，然后用下面的方法来提高我们的估计

$x_1=x_0-\frac{f'(x_0)}{f''(x_0)}$

这样得到一个新的能最小化函数的值，然后将 x_1 替换方程中的 x_0 得到更新后的 x_2。连续进行这样的迭代过程，并可以把它写成一般化的方程

$x_{n+1}=x_n-\frac{f'(x_n)}{f''(x_n)}$

当 $f'(x_n)$ 非常接近 0 的时候，程序停止迭代。通常，需要预先设定一个公差 ε，然后程序在 $|f'(x_n)|<\varepsilon$ 时停止迭代。

可以证明，只要 x_0 足够接近最小值，那么牛顿 -拉夫逊法总能收敛于局部的最小值。与其他数值优化技术相比，当函数存在多个最小值点，牛顿 -拉夫逊法不一定能找到最佳值。不过，如果函数总能满足 $f''(x)>0$，那么就只存在一个最小值。

现实中，使用牛顿 -拉夫逊法常常会遇到一些棘手的问题。比如会遇到 $f''(x_n)=0$ 的情况，这时候函数在这个区段内看起来像一条直线，这样无法求解泰勒近似方程。这种情况下，仅根据 $f'(x_n)$，然后朝着函数值下降的方向，将 x 移动一小步。

另一种情况是，如果 x_n 离真正的最小值太远，泰勒近似的值不太正确以至于 $f(x_{n+1})$ 大于 $f(x_n)$。此时可以用 $(x_{n+1}+x_n)/2$ 来代替 x_{n+1}（或者其他处于 x_n 和 x_{n+1} 之间的值），目的是用小一点的步长来得到更精确一点的结果。

最后，计算 $f'(x)$ 和 $f''(x)$ 的代码可能包含了缺陷，所以应该时时记得检验一下代码。

例 7.1

求函数 $f(x)=e^{-x}+x^4$ 的最小值。经检查函数可知，最小值应该出现在 0 的右边，因为 e^{-x} 是一个减函数，而 x^4 的最小值出现在 $x=0$ 处。可以先画出函数的图来帮助我们猜一个初始值（图 7.3）。

```
f <- function(x) {exp(-x) + x^4}
curve(f, from=-1, to=4)
```

从图中可见，函数的最小值大约出现在 $x_0=0.5$ 的地方，我们就将它设为初始值。由于前面已经提到的困难，所以不会尝试去写一个通用的牛顿 -拉夫逊函数。下面将评估几次迭代过程，看函数是否收敛。

```
f <- function(x) {exp(-x)+x^4}
fprime <- function(x) {-exp(-x)+4*x^3}
fprimeprime <- function(x) {exp(-x)+12*x^2}
x <- c(0.5, rep(NA, 6))
fval <- rep(NA, 7)
fprimeval <- rep(NA, 7)
fprimeprimeval <- rep(NA, 7)
for (i in 1:6) {
+ fval[i] <- f(x[i])
+ fprimeval[i] <- fprime(x[i])
+ fprimeprimeval[i] <- fprimeprime(x[i])
+ x[i+1] <- x[i]-fprimeval[i]/fprimeprimeval[i]
+ }
data.frame(x, fval, fprimeval, fprimeprimeval)
```

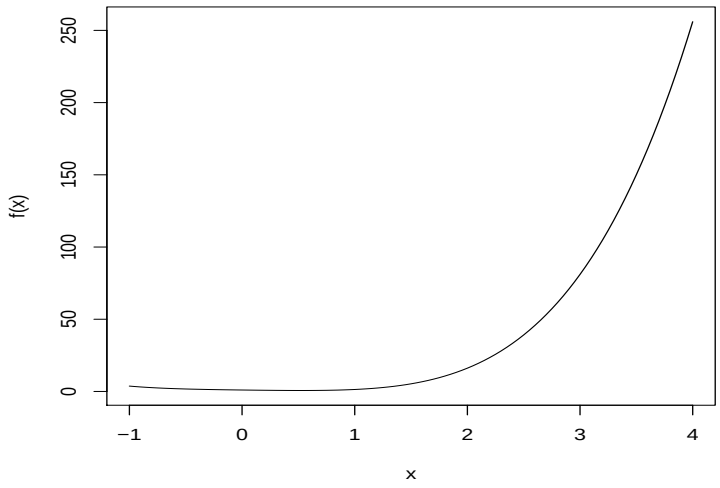


图 7.3: 函数 $f(x) = e^{-x} + x^4$

	x	fval	fprimeval	fprimeprimeval
1	0.5000000	0.6690307	-1.065307e-01	3.606531
2	0.5295383	0.6675070	5.076129e-03	3.953806
3	0.5282544	0.6675038	9.980020e-06	3.938266
4	0.5282519	0.6675038	3.881429e-11	3.938235
5	0.5282519	0.6675038	0.000000e+00	3.938235
6	0.5282519	0.6675038	0.000000e+00	3.938235
7	0.5282519	NA	NA	NA

在上面的结果中可以看到，函数收敛得很快，一阶导数在第四次迭代时就已经等于零了，而此时二阶导数为正，说明此处是一个局部最小值。事实上，二阶导数 $f''(x) = e^{-x} + 12x^2$ 处处为正，所以确定这是一个全局最小值。

7.3 内尔德 -米德单纯形法

前面谈到了两种数值优化的方法，他们都针对于单变量情况。当一个函数有多个输入变量时，优化过程变得非常困难。当变量多于两个后，我们甚至都无法将函数绘画出来。

内尔德 -米德法是对多变量函数最优化处理的算法之一。例如对于一个 p 维的函数，该算法从 $p+1$ 个点开始，$x_1 + x_2 + \cdots + x_{p+1}$，如果能把它们看做一个 p 维的实体（一个单纯性），它们将必然包含一个非零值。例如，在一个两维情况下，这三个点不能都位于同一直线上，所以它们将组成一个三角形。在三维情况下，四个点应该形成一个四面体。

这些点按照 $f(x_i)$ 的值从小到大排序，$f(x_1) \leq f(x_2) \leq \cdots \leq f(x_{p+1})$。这样做的目的是为了最小化 $f(x)$，我们使用的方法是寻找一个更小值的点来替换 x_{p+1}。寻找这个更小值的点，可以通过已知的点计算几个参考点 z_i。图 (7.4) 显示面对二维情况时，有四种方式可以计算这些参考点。前面三种是计算 $x_1, \ldots, x_p$ 之间的中点，计算方法是 $x_{mid} = (x_1 + \cdots + x_p)/p$。

1. 反射法：把 x_{p+1} 通过 x_{mid} 反射到 z_1。

2. 反射并扩展：x_{p+1} 通过 x_{mid} 反射，并且在两倍距离的地方得到 z_2。

3. 收缩：x_{p+1} 通过 x_{mid} 收缩一半距离，得到 z_3。
4. 收缩：所有的点都向着 x_1 的方向收缩一半距离，得到 $z_4, \ldots, z_{p+3}$。

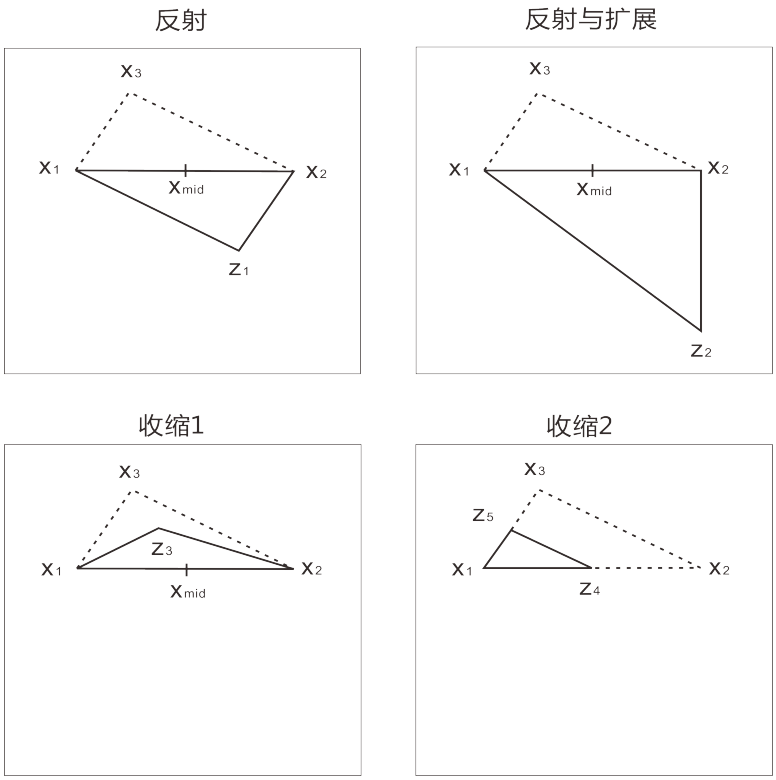


图 7.4: 构建 z_i 点的方法

根据函数 $f(z_i)$ 值的顺序，依次检验这些点。在阅读下面的函数代码时，参照图7.5很有帮助。

```
初始化:
将初始点保存在矩阵 x 中, 这样第 i 个点就是
x[i,]
For i in 1:(p + 1) 计算 f(x[i,])
将这些点重新设定标签
f(x[1,]) <= f(x[2,]) <= ... <= f(x[p + 1,])
计算中点 xmid = (x[1,] + x[2,] + ... + x[p,]) / p
检验:
通过反射法计算 z1: z1 <- xmid - (x[p + 1,] - xmid)
If f(z1) < f(x[1,]) { # 区间 A
通过反射与扩展法计算 z2:
z2 <- xmid - 2 * (x[p + 1,] - xmid)
```

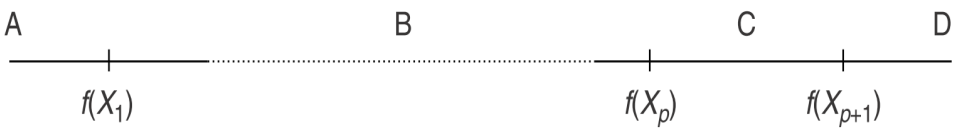


图 7.5: $f(z_i)$ 在内尔德 -米德算法中可能的区间

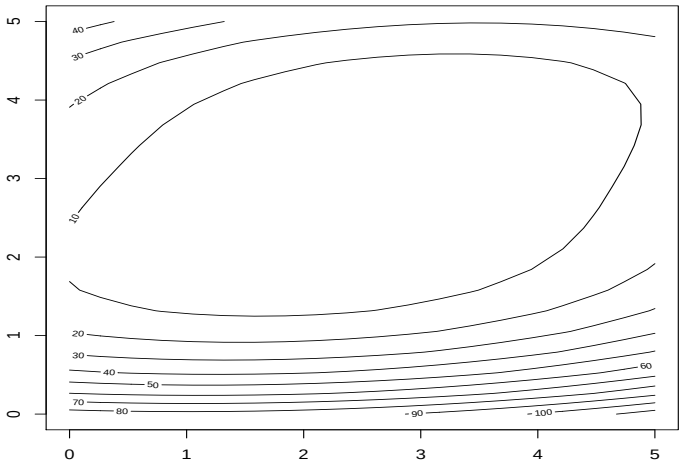


图 7.6: 函数值轮廓线

If f(z2) < f(z1) return(z2)
else return(z1)
} else {
If f(z1) < f(x[p,]) return(z1) # 区间 B
If f(z1) < f(x[p + 1,]) {
Swap z1 with x[p + 1,] # 区间 C
}
}
到这里我们知道 f(z1) 在区间 D.
尝试收缩法 1, 得到 z3.
If f(z3) < f(x[p + 1,]) return(z3) # 区间 A, B, 或 C
所有的都适用, 那么适用收缩法 2 把所有的点都向 x[1,] 移动。

例 7.2

这个例子使用内尔德 -米德求下面函数的最小值：

```
f <- function(x,y){(x-y)^2+(x-2)^2+(y-3)^4}
```

先画出这个函数的轮廓图，以便估计初始值。可以用下面的代码来生成这个图，见图 (7.6)。

```
x <- seq(0,5,len=20)
y <- seq(0,5,len=20)
z <- outer(x,y,f)
contour(x,y,z)
```

用 neldermead(x,f) 函数来作此计算，其中 x 是前面列出代码中的矩阵，f 是需要计算的函数。函数 neldermead(x,f) 的返回值是经过更新后的矩阵 x。下面的日志文件显示了 9 次详细的更新过程。图是每一步计算过程的示范。

```
> x <- matrix(c(0, 0, 2, 0, 2, 0), 3, 2)
> polygon(x)
```

```
> for (i in 1:9) {
+ cat(i,":") + x <- neldermead(x,f) + polygon(x) +
text(rbind(apply(x, 2, mean)), labels=i) + }
1 :Accepted reflection , f(z1)= 3.3
2 :Swap z1 and x3
Accepted contraction 1, f(z3)= 3.25
3 :Accepted reflection and expansion , f(z2)= 0.31875
4 :Accepted reflection , f(z1)= 0.21875
5 :Accepted contraction 1, f(z3)= 0.21875
6 :Accepted contraction 1, f(z3)= 0.1
7 :Accepted contraction 1, f(z3)= 0.04963379
8 :Accepted contraction 1, f(z3)= 0.03874979
9 :Swap z1 and x3
Accepted contraction 1, f(z3)= 0.02552485
> x
```

```
          [,1]     [,2]
[1,] 2.609375 2.656250
[2,] 1.937500 2.625000
[3,] 2.410156 2.460938
```

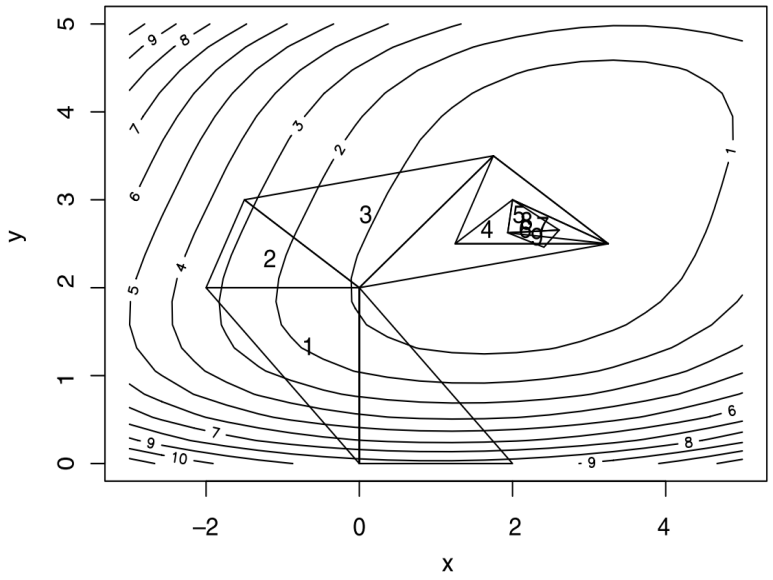


图 7.7: 内尔德 -米德算法更新过程

经过 9 此迭代更新后，可以看到 x 的值大约处于 $1.9-2.6$ 之间，而 y 大约在 $2.4-2.7$ 之间。再经过 50 次迭代后，函数收敛到真正的最小值附近 $(x,y)=(2.25,2.5)$。

7.4 内建函数

R 之中有好几个通用的优化函数。在一维情况下，optimize() 函数采用同属黄金分割法的一种变种算法。函数 optim() 则可用于多维情况，它包含了几种不同的优化算法，包括内尔德 -米德法，牛顿 -拉夫逊法以及其他一些我们没提到的算法。

引用该函数语法为：optim(par, fn, ...)。其中 *par* 设定了 optim() 中的初始参数，除了告诉 optim() 从何处开始，还包括 fn 中包括多少变量。第二个变量 fn 是需要最小化的目标函数。它的第一个参数应该是和

par 长度一样的向量，optim() 函数不断调用这个参数以寻找最小值。最后函数会返回一个数值。还有一些可选参数可参照帮助文件。

R 中还有一些通用的优化函数，比如 nlm() 和 nlminb()。在大部分情况下，我们推荐用 optim()，因为它的适用性更强。constrOptim() 则可用于参数中包含线性不等式约束的情况。

习题

1. 用 optimize() 函数求下面方程的最小值：

   (a) $f(x) = |x - 3.5| + |x - 2| + |x - 1|$

   (b) $f(x) = |x - 3.2| + |x - 3.5| + |x - 2| + |x - 1|$

2. 用 nlm() 和 optim() 函数求下面方程的最小值，

   $f(a, b) = (a - 1) + 3.2/b + 3\log(\Gamma(a) + 3a\log(b))$

   其中 $\Gamma(a)$ 是伽马函数，可以直接在 R 中调用 $gamma(a)$。

3. 用 nlminb() 函数重新计算上面一题，其中 a 和 b 取非负值。

7.5 线性规划

现实中经常要面对在约束条件下最小化（或最大化）一个函数。当函数是线性的，并且约束条件能被表达为线性等式或不等式时，这一类问题就被称做线性规划。

这一类问题的标准形式可表达为：

$$\min_{x_1,x_2,\ldots,x_k} C(x) = c_1x_1 + \cdots + c_kx_k$$

约束条件为:

$a_{11}x_1 + \cdots + a_{1k}x_k \geq b_1$

$a_{21}x_1 + \cdots + a_{2k}x_k \geq b_2$

$\cdots$

$a_{m1}x_1 + \cdots + a_{mk}x_k \geq b_m$

以及满足非负条件 $x_1 \geq 0, \ldots, x_k \geq 0$。

线性规划的意图是在约束条件以及非负条件下，寻找能最小化目标函数 $C(x)$ 的决策变量 $x_1, x_2, \ldots, x_n$。

例 7.3

某工厂设计了两种工艺来降低二氧化硫和二氧化碳的排放。第一种工艺减少同等数量的废气排放，每单位的废气需要花费 5 美元，第二种工艺可以减少同等数量的二氧化硫，但可以减少两倍的二氧化碳，每单位成本是 8 美元。

假设该工程需要减少排放 2 百万单位的二氧化硫，3 百万单位的二氧化碳，那么应该怎么选择这两种工艺能使成本最小？

令 x_1 代表用第一种工艺处理的废气总量，x_2 代表使用第二种工艺处理的废气总量，假设以百万位单位。那么总成本可以表达为（百万美元）：

$C = 5x_1 + 8x_2$

因为两种工艺降低二氧化硫的效率相同，所以减少二氧化硫总量为: $x_1 + x_2$。因为该工厂需要减少 2 百万单位的二氧化硫，所以：

$x_1 + x_2 \geq 2.$

第二种工艺减少二氧化碳的效率是第一种工艺的两倍，并且该工厂需要减少排放 3 百万单位的二氧化碳，于是第二个约束条件为：

$x_1 + 2x_2 \geq 3$

最后，还要注意到，x_1 和 x_2 不能小于零。这样，我们得到线性规划问题：

$\min C = 5x_1 + 8x_2$

s.t.

$x_1 + x_2 \geq 2.$

$x_1 + 2x_2 \geq 3$.

$x_1, x_2 \geq 0$

上述关系可以用图 (7.8) 来表示。其中的灰色阴影区域是*可能区域*，这是所有满足约束条件的 (x_1, x_2) 的组合。白色区域是不满足约束条件的区域。

函数 $C(x)$ 的梯度是 $(5, 8)$，这个向量给出了函数值下降最快的方向，函数的轮廓线应该垂直于这个向量。所有轮廓线中的一条我们用灰色虚线画在图 (7.8) 中。函数最小化的解位于第一条与可能区域相交的轮廓线上。如果交点只有一个，那么有唯一的最小值。在本例中，这个交点位于 $(1, 1)$。

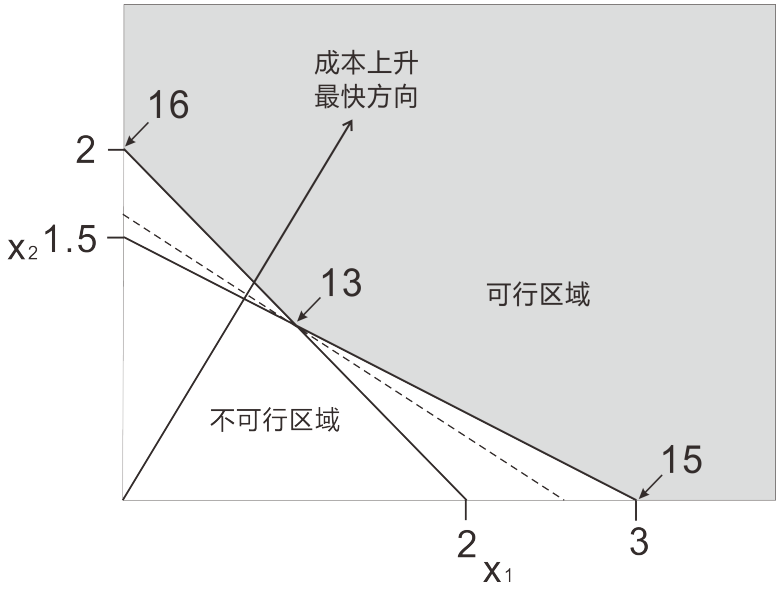


图 7.8: 线性规划的约束域

7.5.1 求解线性规划问题

虽然在 R 中有多个可用于线性规划的函数，但当前来看，在 lpSolve 宏包中提供的 lp() 函数最稳定。该函数基于改进的单纯形法，这种方法在可能域中很智能地测试一些极端点，看它们是否是最优解。

lp() 函数中包括若干参数，下面列出的几个在求解最小化问题时可能会用得着：

- objective.in: 目标函数的系数向量。
- const.mat: 约束条件左侧决策变量的系数矩阵，每一行代表一个约束条件。
- const.dir：一个字符串向量，表明约束条件不等式的方向，可取的值包括 >=, == 和 <=。
- const.rhs：约束条件右侧的常数向量。

例 7.4

用本节介绍的函数解例 7.3 中的最小化问题：

```
library(lpSolve)
eg.lp <- lp(objective.in=c(5,8), const.mat=matrix(c(1,1,1,2),nrow=2),
+ const.rhs=c(2,3), const.dir=c(">=", ">="))
eg.lp
```

Success: the objective function is 13

```
eg.lp$solution
```

```
[1] 1 1
```

输出结果显示，目标函数的最小值是 13，取得最小值的位置是 $x_1 = 1, x_2 = 1$。

7.5.2 最大化及其他约束

当把参数设成 direction = "max"，lp() 函数就能用于处理最大化问题。

例 7.5

求解下面的最大化问题：
$\max C = 5x_1 + 8x_2$
s.t.
$x_1 + x_2 \leq 2.$
$x_1 + 2x_2 = 3.$
$x_1, x_2 \geq 0$

在 R 中，处理这样的问题很简单，请看下面的代码：

```
eg.lp <- lp(objective.in=c(5,8),
+ const.mat=matrix(c(1,1,1,2), nrow=2),
+ const.rhs=c(2,3),
+ const.dir=c("<=", "="), direction="max")
eg.lp$solution
```

```
[1] 1 1
```

结果显示，在点 $(1, 1)$ 处取得极大值 13。

7.5.3 特殊情况

多重最优解
有时候在一个线性规划问题中会出现多个最优解。

例 7.6

将例 7.3 稍作修改
$\min C = 4x_1 + 8x_2$
s.t.
$x_1 + x_2 \geq 2.$
$x_1 + 2x_2 \geq 3.$
$x_1, x_2 \geq 0$
这个问题有两个最优解，分别在 $(1, 1)$ 和 $(3, 0)$。连接这两个点的线条上的所有点也是最优解。图 (7.9) 显示了这个解的集合。
不过函数 lp() 并不会警告用户存在多个最优解，事实上本例在 R 中运行后的输出结果是 $x_1 = 3, x_2 = 0$。
简并
假设一个问题包含 m 个决策变量，那么当某一个点面对的约束超过 m 时，就出现简并情况。这种情况非常罕见的，但它有可能对使用单纯形法造成困难，因此, 要特别注意这个条件。在非常罕见的情况下，简并可以阻挠函数收敛到最优解，不过大多数时候不需要太担心。

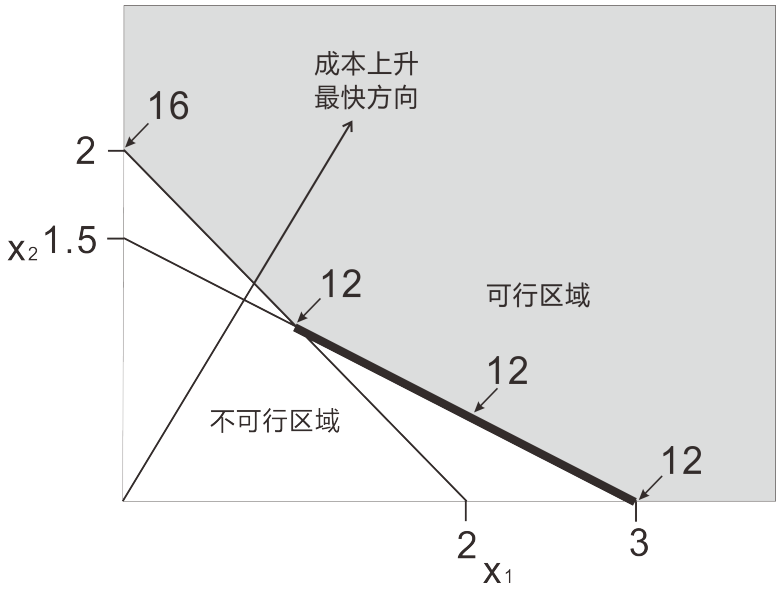


图 7.9: 多个最优解

例 7.7

下面的例题中包含一个简并点，但这个简并点不是在最优解的位置。不过使用 lp() 函数还是毫无困难的找到了最优解。

$\min C = 3x_1 + x_2$
s.t.
$x_1 + x_2 \geq 2.$
$x_1 + 2x_2 \geq 3.$
$x_1 + 3x_2 \geq 4.$
$4x_1 + x_2 \geq 4.$
$x_1, x_2 \geq 0$
约束条件的边界显示在图 (7.10) 中，此题可以很容易的求解：

```
degen.lp <- lp(objective.in=c(3,1),
+ const.mat=matrix(c(1,1,1,4,1,2,3,1), nrow=4),
+ const.rhs=c(2,3,4,4), const.dir=rep(">=",4))
degen.lp
```

Success: the objective function is 3.333333

```
degen.lp$solution
```

[1] 0.6666667 1.3333333

不可能性
不可能性是一个很常见的问题。当约束条件不能同时成立时，就不存在可能的解。

例 7.8

$\min C = 5x_1 + 8x_2$
s.t.

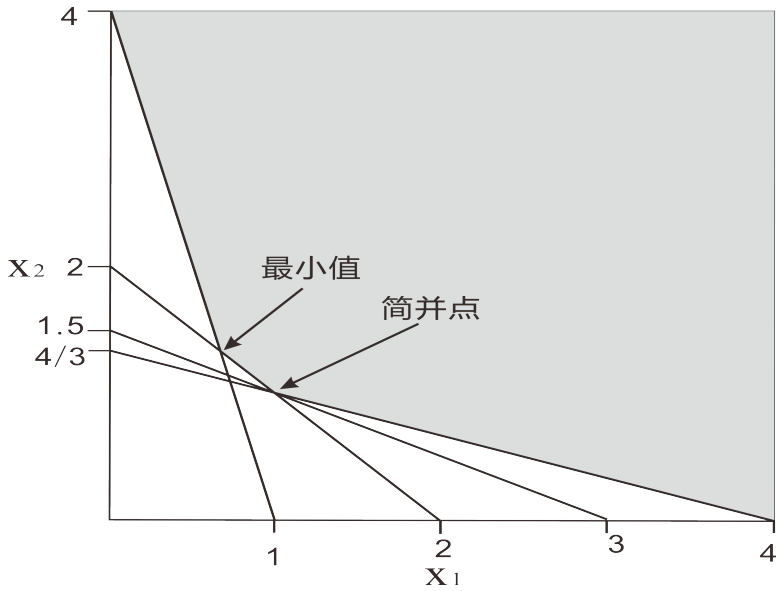


图 7.10: 约束条件简并

$x_1 + x_2 \geq 2.$
$x_1 + x_2 \leq 1.$
$x_1, x_2 \geq 0$
如果将上述约束条件设为 lp() 函数的参数，那么会返回下面这样的出错信息：

```
eg.lp <- lp(objective.in=c(5,8),
+ const.mat=matrix(c(1,1,1,1), nrow=2),
+ const.rhs=c(2,1), const.dir=c(">=", "<="))
eg.lp
```

Error: no feasible solution found

无边界性
有时候，约束条件和目标函数给出了无边界的解。

例 7.9

$\max C = 5x_1 + 8x_2$
s.t.
$x_1 + x_2 \geq 2.$
$x_1 + 2x_2 \leq 3.$
$x_1, x_2 \geq 0$
这个问题的可行解区域和例 7.3 是一样的，可参阅图 7.8。但是现在不是最小化目标函数，而是求目标函数的最大值，所以需要沿着增大目标函数的方向搜寻。此时我们发现，只要取任意大的 x_1 和 x_2，可以无限增大目标函数的值。应用 lp() 来解这个题时，就会返回下面的出错信息。

```
eg.lp <- lp(objective.in=c(5,8),
+ const.mat=matrix(c(1,1,1,2), nrow=2),
+ const.rhs=c(2,3), const.dir=c(">=", ">="),
+ direction="max")
eg.lp
```

Error: status 3

当约束条件和/或目标函数没有正确公式表达的时候，容易出现无边界性的情况。

7.5.4 非约束变量

有时候决策变量并没有限定为非负。lp() 函数并不能直接用于处理此类问题，不过用一个简单的方法就可处理。

如果 x 没有限定符号，那么 x 可以写成 $x_1 - x_2$，其中 $x_1 \geq 0, x_2, \geq 0$。这意味着任何一个未约束变量可以用两个非负变量的差来表示。

例 7.10

$\min C = x_1 + 10x_2$
s.t.
$x_1 + x_2 \geq 2.$
$x_1 - x_2 \leq 3.$
$x_1 \geq 0$
我们注意到 x_2 并没有限制符号，所以将其改写为 $x_2 = x_3 - x_4$，将它代入原方程的得：
$\min C = x_1 + 10x_3 - 10x_4$
s.t.
$x_1 + x_3 - x_4 \geq 2.$
$x_1 - x_3 + x_4 \leq 3.$
$x_1 \geq 0, x_3 \geq 0, x_4 \geq 0$
对应的 R 程序为，

```
unres.lp <- lp(objective.in=c(1, 10, -10),
+ const.mat=matrix(c(1, 1, 1, -1, -1, 1), nrow=2),
+ const.rhs=c(2, 3), const.dir=c(">=", "<="))
unres.lp
```

Success: the objective function is -2.5

```
unres.lp$solution
```

[1] 2.5 0.0 0.5

所得的解是 $x_1 = 2.5$，$x_2 = x_3 - x_4 = 0.0 - 0.5 = -0.5$。

7.5.5 整数规划

决策变量常常会被限制为只能取整数。例如，要选择用一辆，两辆或三辆卡车来运货，以便于最小化成本支出。这时候就只能使用整数辆的卡车。这一类包含整数变量的决策问题称做整数规划。面对这类问题时，简单的把最后结果四舍五入到整数并不是一个很好的方法，有时候结果会偏离最优解很远。

在 R 中，lp() 函数通过使用一种称为分支界定算法可以很容易处理整数变量问题。通过 int.vec 这个参数可以指定哪些变量限制为整数。

例 7.11

寻找合适的 x_1, x_2, x_3 和 x_4，最小化目标函数
$C(x) = 2x_1 + 3x_2 + 4x_3 - x_4$
s.t.
$x_1 + 2x_2 \geq 9$
$3x_2 + x_3 \geq 9$
$x_2 + x_4 \leq 10$
其中 x_2, x_4 只能取整数。这类问题的 R 代码为：

```
integ.lp <- lp(objective.in=c(2,3,4,-1),
+ const.mat=matrix(c(1,0,0,2,3,1,0,1,0,0,0,1), nrow=3),
+ const.dir=c(">=", ">=", "<="), const.rhs=c(9,9,10),
+ int.vec=c(2,4))
integ.lp
```

Success: the objective function is 8

```
integ.lp$solution
```

[1] 1 4 0 6

这样，当 x_2, x_4 限定为整数后，得到的最优解是 $x_1 = 1, x_2 = 4, x_3 = 0$ 和 $x_4 = 6$。假如去掉这一个约束后，结果就不一样了：

```
wrong.lp <- lp(objective.in=c(2,3,4,-1),
+ const.mat=matrix(c(1,0,0,2,3,1,0,1,0,0,0,1), nrow=3),
+ const.dir=c(">=", ">=", "<="), const.rhs=c(9,9,10))
wrong.lp
```

Success: the objective function is 8

```
wrong.lp$solution
```

[1] 0.0 4.5 0.0 5.5

把上述结果取最接近的整数并不合适，例如，如果 x_2 取 4，则会违反第一条约束条件，如果 x_2 取 5，那么目标函数的最小值就会大于 8。

7.5.6 LP() 外的其他选择

lp() 函数提供了一个与 C 语言相连的接口，在 linpro 宏包中还有一个完全用 R 语言编写的函数 solveLP()。后面这个函数在求解较大的问题时速度很慢，但结果更可靠些。我们还要指出 boot 宏包中的函数 simplex()，在面对特别大的问题时，单纯形法可能不会很快收敛，此时有一种更好的算法叫做内点法 (interior point method) 可供选用，不过这种方法目前还没有对应的 R 程序。

7.5.7 二次规划

在约束条件下，最小化一个非线性目标函数时，线性规划可能只是其中的一个特殊解。关于非线性函数，通常更难求解，也超出了本书的范围。不过目标函数是二次函数，约束条件是线性约束则是一个例外。这一类问题叫做二次规划。

一个包含 k 个约束条件的二次规划问题，通常可以表达为：

$$\min_{\beta} \frac{1}{2}\beta^T D\beta - d^T\beta$$

约束条件为 $A^T\beta \geq b$。其中 β 是包含 p 个未知值的向量，D 是一个 $p \times p$ 的正定矩阵，d 是一个长为 p 的向量，A 是一个 $p \times k$ 的矩阵，b 是一个长度为 k 的向量。

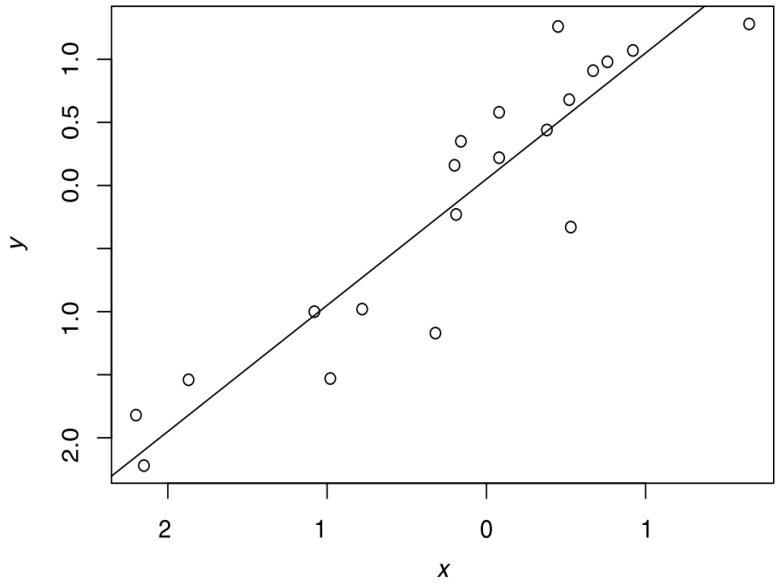


图 7.11: x-y 散点图

例 7.12

假设有下面 20 对观察值，数据结构显示在图 (7.11) 中，

```
x <- c(0.45, 0.08, -1.08, 0.92, 1.65, 0.53, 0.52,
+ -2.15, -2.20, -0.32, -1.87, -0.16, -0.19, -0.98,
+ -0.20, 0.67, 0.08, 0.38, 0.76, -0.78)
y <- c(1.26, 0.58, -1.00, 1.07, 1.28, -0.33, 0.68,
+ -2.22, -1.82, -1.17, -1.54, 0.35, -0.23, -1.53,
+ 0.16, 0.91, 0.22, 0.44, 0.98, -0.98)
```

假设要寻找一条穿过这些数据点的“最佳”直线，则这条直线可以用这样的函数表示 $y = \beta_0 + \beta_1 x$，其中 β_0 是在 y 轴上的截距，β_1 是斜率。不过还有一些背景信息显示，斜率 β_1 至少是 1。

可以找到一条直线，应该满足从观察值到直线上相应点的距离平方和最小，即：

$$\min_{\beta_0,\beta_1} \sum_{i=1}^{20} (y_i - \beta_0 - \beta_1 x_i)^2, \text{ s.t. } \beta_1 \geq 1$$

这是一个约束条件下的最小二乘法问题，它等价于：

$\min_\beta \beta^T X^T X \beta - 2 y^T X \beta$，s.b. $A\beta \geq b$。

其中 $A = [0, 1]$，$\beta = [\beta_0, \beta_1]^T$，$y$ 是一个 20 行的向量，包括观察数据中的 20 个 y 值，X 是一个两列的矩阵，其中第一列的元素都是 1，第二列包括观察数据中的 x 值：

$$X = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \cdots & \cdots \\ 1 & x_n \end{bmatrix} = \begin{bmatrix} 1 & 0.45 \\ 1 & 0.08 \\ \cdots & \cdots \\ 1 & -0.78 \end{bmatrix}$$

于是，我们有：

$$X^T X = \begin{bmatrix} n & \sum_{i=1}^n x_i \\ \sum_{i=1}^n x_i & \sum_{i=1}^n x_i^2 \end{bmatrix} = \begin{bmatrix} 20 & -3.89 \\ -3.89 & 21.4 \end{bmatrix}$$

$$y^T X = [\sum_{i=1}^n y_i, \sum_{i=1}^n x_i y_i] = [-2.89, 20.7585]$$

这是一个二次规划问题，其中 $D = X^T X$，$d = y^T X$。

线性规划方法已经修改适用于二次规划问题了。在 quadprog 宏包中的 solve.QP() 函数就可用于求解二次规划问题，它包括下面这些参数：

- Dmat：它是一个矩阵，包含目标函数的二次型。
- dved: 包含目标函数中决策变量系数的向量。
- Amat：约束条件中决策变量系数的矩阵，矩阵的每一行代表一个约束条件。
- bvec: 包含约束条件右侧常数值的向量。
- mvec: 一个数字，指明约束条件中等式的数量。缺省值是 0。如果不等于 0，那么要将等式约束条件排在非等式约束条件之前。

这个函数的输出是一个列表，其中前两个元素分别是能最小化目标函数的系数，以及目标函数取得的最小值。

例 7.13

在例 7.12 这个约束条件下最小二乘法问题中，设定各个参数：$D = X^T X$，$d = X^T y$，以及 A 和 b。

```
library(quadprog)
X <- cbind(rep(1, 20), x)
XX <- t(X) %*% X
Xy <- t(X) %*% y
A <- matrix(c(0, 1), ncol=1)
b <- 1
solve.QP(Dmat=XX, dvec=Xy, Amat=A, bvec=b)
```

```
$solution

[1] 0.05 1.00

$value

[1] -10.08095

$unconstrainted.solution

[1] 0.04574141 0.97810494


$iterations

[1] 2 0


$iact

[1] 1
```

按照运算结果，可知所要求的直线应该是 $\hat{y} = 0.05 + x$。

运算结果的其他部分显示，约束条件是有效的。如果在未约束条件下，求得的斜率也大于 1，那么约束条件就无效了，此时非约束条件下的结果应该和约束条件下的结果一致。

大家应该注意到，上面的示例中决策变量有符号限制。如果所解问题需要，在使用函数 solve.QP() 时，应该明确设定非负条件。还要注意到，默认的非等式约束都是“大于或等于”，假如面临的问题包括了“小于或等于”的约束条件，那么应该将该约束乘以“(-1)”以符合函数的要求。

应该指出，在其他科学计算环境下，有效率更高的方法来解约束条件下的最小二乘法问题。在前面的例子中，矩阵 D 是一个对角矩阵，采用这种特殊结构可减少计算量。下面的例子中采用了完整的矩阵，而且还限制了决策变量的符号。

例 7.14

二次规划方法可以用于最佳投资组合问题，投资者要做出决策在一个股票集合中对每个股票各投资多少钱。这个问题可以简化为下面的模型：

$\max x^T\beta - \frac{k}{2}\beta^T D\beta$，s.t. $\sum_{i=1}^{n}\beta_i = 1, \beta_i \geq 0$，for $i = 1, \ldots, n$。

在 β 向量中第 i 个元素表示投资者购买第 i 个股票所花经费占总经费的百比例，所以这些元素必须是非负的。向量 x 指每个股票的日均回报（一个股票的日均回报是指该股票当日收盘价与上一日收盘价的差）。所以 $x^T\beta$ 就相当于投资者每天得到的投资回报。

大部分的投资者不愿意承担很大的风险，目标函数中的第二部分就考虑了这个风险问题。参数 k 代表了投资者对风险的忍耐力。如果投资者只想获得最大的投资回报而不在乎风险，那么 $k = 0$。投资者选择的 k 越大，他忍受的风险越小。矩阵 D 称做协方差矩阵，它是对投资回报不确定性的量化。矩阵对角线上的元素表示每个股票投资回报的方差。非对角线上的元素，第 (i, j) 个元素表示第 i 个股票和第 j 个股票投资回报的协方差，这是表示两者回报之间差别的一个简单度量方式。

假设某一个投资者购买三种股票，他本人的风险系数是 $k = 4$，投资回报的协方差矩阵为

$$D = \begin{bmatrix} 0.010 & 0.002 & 0.002 \\ 0.002 & 0.010 & 0.002 \\ 0.002 & 0.002 & 0.010 \end{bmatrix}$$

假设这三个股票的日均回报是 $0.002, 0.005$ 和 0.01，即 $x^T = [0.002, 0.005, 0.01]$

对于约束条件中 $\beta_1 + \beta_2 + \beta_3 = 1$ 和所有 β 值都非负的要求，可以用下面的矩阵来表示：

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} \begin{matrix} = \\ \geq \\ \geq \\ \geq \end{matrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

所以设定

$$A^T = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

最大化一个目标函数等价于最小化目标函数的负值，所以可以用 solve.QP() 函数来解这个问题。

```
A <- cbind(rep(1, 3), diag(rep(1, 3)))
D <- matrix(c(0.01, 0.002, 0.002, 0.002, 0.01,
+0.002, 0.002, 0.002, 0.01), nrow=3)
x <- c(0.002, 0.005, 0.01)
b <- c(1, 0, 0, 0)
solve.QP(2 * D, x, A, b, meq=1)
```

```
$solution

[1] 0.1041667 0.2916667 0.6041667


$value

[1] -0.002020833


$unconstrainted.solution

[1] -0.02678571 0.16071429 0.47321429


$iterations

[1] 2 0
```

```
$iact
[1] 1
```

对于这个投资者来说，最佳的投资策略是，将 10.4% 的资金投资于第一个股票，将 29.2% 的资金投资于第二个股票，将 60.4% 的资金投资于第三个股票。最佳的投资收益为 0.0020 （显示在 $value 中，因为模型中最小化目标函数的负值，所以上面显示的值是负的。）。

习题

1. 寻找合适的非负的 x_i 来最小化目标函数
   $C(x) = x_1 + 3x_2 + 4x_3 + x_4$，其中约束条件为：
   $x_1 - 2x_2 \geq 9$
   $3x_2 + x_3 \geq 9$
   $x_2 + x_4 \geq 10$
2. 如果上题中某一个 x 被限制为只能取整数值，结果会有变化吗？请解释
3. 假设上题中目标函数是 $C(x) = x_1 - 3x_2 + 4x_3 + x_4$，结果有何改变？
4. 寻找合适的非负的 x_i 来最小化目标函数
   $C(x) = x_1 + 3x_2 + 4x_3 + x_4$，其中约束条件为：
   $x_1 - 2x_2 \leq 9$
   $3x_2 + x_3 \leq 9$
   $x_2 + x_4 \leq 10$

7.6 本章习题

1. 使用例 7.12 中的数据，假设约束条件要求截距不小于斜率，这条最佳直线又是如何？
2. 在投资组合的例子中，如果投资者风险系数 $k = 1$，投资策略会有什么改变？
3. 通常，投资于某一个股票的份额会有限制。那么在投资组合的例子中，如果投资于任何一个股票的资金不能超过 50%，新的投资策略会是什么？
4. 有两个公司的股票，DDI 公司和 JJL 公司，其中 DDI 公司股票的日均回报率是 0.005，JJL 公司股票的投资回报率是 0.010。假如某个投资者是风险爱好者，即 $k = 0$。那么他的投资策略应该是什么？（提示：本题不需要计算）
5. 假设 DDI 和 JJL 公司股票的投资回报是独立的，但 $\sigma^2_{DDI} = 0.01$，$\sigma^2_{JJL} = 0.04$。那么当投资者的风险系数分别为 (a) $k = 1$, (b) $k = 2$ 时，投资组合分别是多少？
   已知 $D = \begin{bmatrix} 0.01 & 0 \\ 0 & 0.04 \end{bmatrix}$
6. 假如上面的题目中，两个股票的收益有关联的话，又会怎样。
   已知 $D = \begin{bmatrix} 0.01 & 0.01 \\ 0.01 & 0.04 \end{bmatrix}$

第 八 章

部分习题答案

第二章

2.2

1. > 170166719 %% 31079

   [1] 9194

3. (a) > r <- 1.08

   > n <- c(10, 20, 30, 40)

   > sum1 <- c()

   > for(i in n){

   + x <- 1:i

   + sum1 <- c(sum1, sum(r^x))

   + }

   > sum1

   [1] 15.64549 49.42292 122.34587 279.78104

   这里分别给出了 n = 10, 20, 30, 40 时的和。

   > sum2 <- (1 - r^(n + 1)) / (1 - r)

   > sum2

   [1] 16.64549 50.42292 123.34587 280.78104

   > sum2 - sum1

   [1] 1 1 1 1

   这个公式是实际的和再加一.

5. > Sum <- function(n){

   + x <- 1:n

   + ans <- sum(x)

   + ans

   + }

   (a) > n <- 100

   > Sum(n)

```
[1] 5050
> formula <- (n * (n + 1)) / 2
> Sum(n) - formula
[1]  0
```

(b)
```
> n <- 400
> formula <- (n * (n + 1)) / 2
> Sum(n) - formula
[1] 0
```

9.
```
> Sums <- function(N){
+ x <- 1:N
+ y <- (1/x)
+ sums <- sum(y)
+ return(sums)
+ }
> formula <- function(N){
+ log(N) + 0.6
+ }
```

(a)
```
> N <- 500
> Sums(N) - formula(N)
[1] -0.02178467
```

(c)
```
> N <- 2000
> Sums(N) - formula(N)
[1] -0.02253436
```

(e)
```
> N <- 8000
> Sums(N) - formula(N)
[1] -0.02272184
```

10. R 能存储 15 到 16 位的数字，也就是说，诸如 $x[0.9999999999999999]$ 这样的数字， R 能够识别出它小于 1 ，当它用于索引时，会被截断为 0 。它的结果与 $x[0]$ 是一样的，输出结果是 $numeric(0)$ 。然而，对于 $x[0.99999999999999999]$ 这样的数字，因为它有 17 个 9 ，所以 R 将它看做 1 ，所以输出结果是 $x[1]$

11. (a)
```
> rep(0:4,rep(5,5))
[1] 0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4
```

13.
```
> rep(seq(1,5), times = 5) + rep(0:4, each = 5)
```

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

2.3

1 (a) > solar.radiation <- c(11.1,10.6,6.3,8.8,10.7,11.2,8.9,12.2)

(b) > mean(solar.radiation)

[1] 9.975

> median(solar.radiation)

[1] 10.65

> var(solar.radiation)

[1] 3.525

(c) i. > sr10 <- solar.radiation + 10

ii. > mean(sr10)

[1] 19.975

> median(sr10)

[1] 20.65

> var(sr10)

[1] 3.525

iii. 均值和中位数增加 10, 方差不变。

2 方法 1: 可以用 $?var$ 或者 $help(var)$ 查看关于 $var()$ 函数的详细帮助。结果发现计算方差时分母用的是 $(n-1)$

方法 2: 可以手工计算 $solar.radiation$ 的方差，然后和用 $var()$ 计算的结果比较。

> var1 <-(1/length(solar.radiation))

+ *sum((solar.radiation-mean(solar.radiation))^2)

> var2 <-(1/(length(solar.radiation)-1))

+ *sum((solar.radiation-mean(solar.radiation))^2)

> var1

[1] 3.084375

> var2

[1] 3.525

> var(solar.radiation)

[1] 3.525

这可以证明，在 R 中计算方差时分母用的是 $(n-1)$ 。

2.4

2.4.1

1 xor 运算的真值表

A	B	xor
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

3 $!(A\&B) == (!A)|(!B)$

A 和 B 同时不为真的事件等价于要么 A 不为真要么 B 不为真的事件。

``没有太阳雨" 这句话等价于 ``要么天空不晴朗，要么天空不下雨"。

$!(A\&B) == (!A)|(!B)$ 的真值表：

A	B	not(A and B)	(not A) or (not B)
TRUE	TRUE	FALSE	FALSE
TRUE	FALSE	TRUE	TRUE
FALSE	TRUE	TRUE	TRUE
FALSE	FALSE	TRUE	TRUE

本表可证明上述陈述确实为真。

2.4.2

1. 当 A 为假时， B 的值需要通过 $A||B$ 来确定，因为当 A 为真时，我们已经知道 $A||B$ 为真，然而当 A 为假时，那么 $A||B$ 的值就取决于 B 的值了。

3. > b * a

   [1] 13 0 0 2

   b 与 a 中对应的元素两两相乘。

2.5

2.5.2 DUMP() AND SOURCE()

3. > numbers <- c(3,5,8,10,12)

   > dump("numbers", file = "numbers.R")

   > rm(numbers)

   > ls()

   在结果中不会显示数字

```
 [1] "a"          "ans"           "b"             "formula"
 [5] "i"          "intRate"       "more.colours"  "n"
 [9] "N"          "payment"       "principal"     "r"
[13] "randomdata" "solar.radiation" "sr10"        "srm2"
[17] "Sum"        "sum1"          "sum2"          "Sums"
[21] "Sumsqrd"    "values"        "var1"          "var2"
[25] "vp"         "vp1"           "x"
```

> source(”numbers.R”)

> ls()

结果中显示数字

[1]	"a"	"ans"	"b"	"formula"
[5]	"i"	"intRate"	"more.colours"	"n"
[9]	N	"numbers"	"payment"	"principal"
[13]	"r"	"randomdata"	"solar.radiation"	"sr10"
[17]	"srm2"	"Sum"	"sum1"	"sum2"
[21]	"Sums"	"Sumsqrd"	"values"	"var1"
[25]	var2	"vp"	"vp1"	"x"

2.5.6

1. 创建 pretend.df:

> x <- c(61,175,111,124)

> y <- c(13,21,24,23)

> z <- c(4,18,14,18)

> pretend.df <- cbind(x,y,z)

> pretend.df <- data.frame(pretend.df)

> pretend.df

	x	y	z
1	61	13	4
2	175	21	18
3	111	24	14
4	124	23	18

显示：

> pretend.df[1,3]

[1] 4

第二章习题

1. 从所给的网页上下载 $rnf6080.dat$ 文件，并保存到电脑上。

> rain.df <-read.table(”rnf6080.dat”, header = FALSE,
+ na.strings = ”-999”)

(a) > rain.df[2, 4]

[1] 0

(b) > names(rain.df)

[1] ”V1” ”V2” ”V3” ”V4” ”V5” ”V6” ”V7” ”V8” ”V9” ”V10” ”V11” ”V12”

[13]”V13” ”V14” ”V15” ”V16” ”V17” ”V18” ”V19” ”V20” ”V21” ”V22” ”V23” ”V24”

[25]”V25” ”V26” ”V27”

(c) > rain.df[2,]

V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15 V16 V17 V18 V19 V20 V21

2 60 4 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

V22 V23 V24 V25 V26 V27

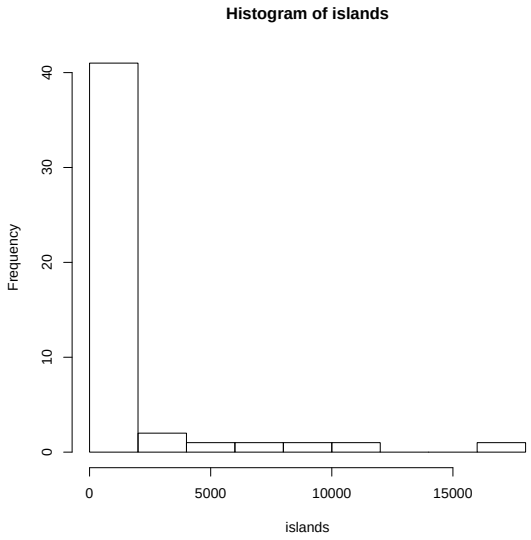
2 0 0 0 0 0 0

(e) > rain.df$daily <- rowSums(rain.df[, 4:27], na.rm=TRUE)

第三章

3.1

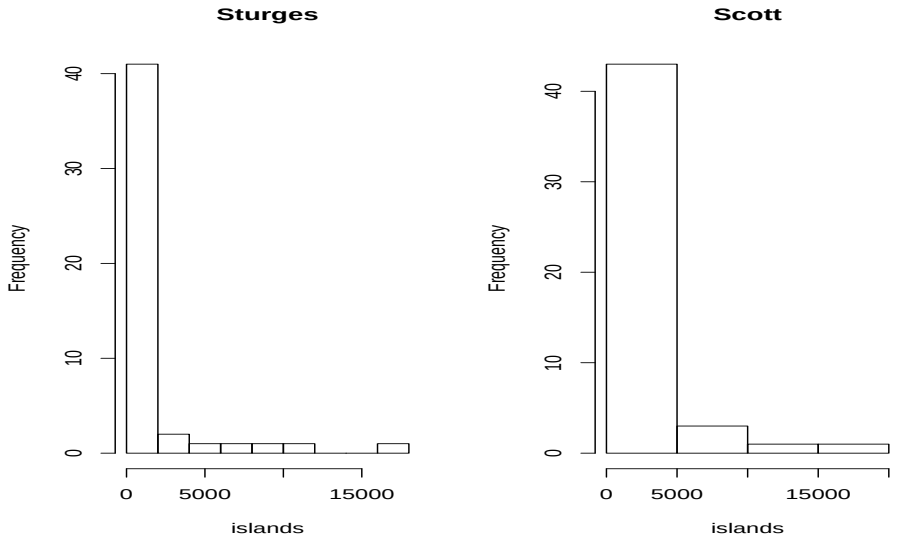
1. (a) > hist(islands)



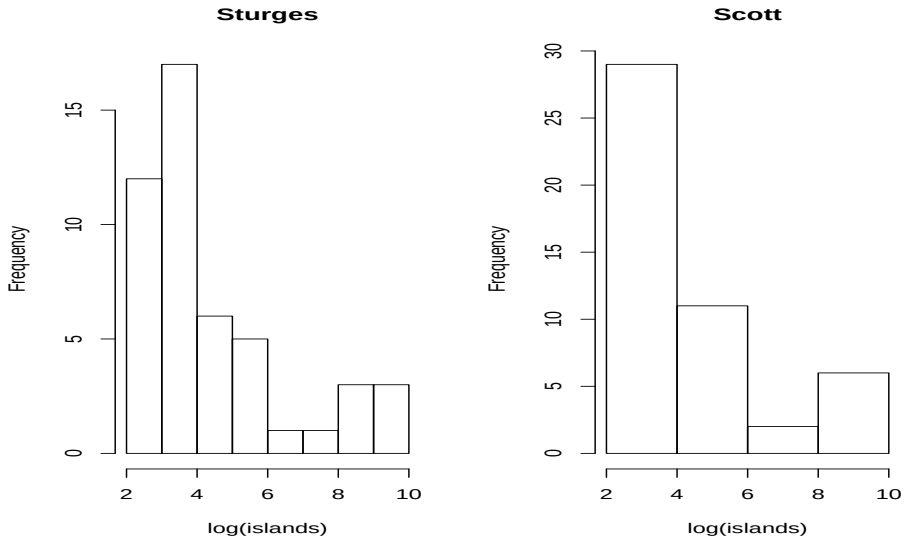
(c) > par(mfrow=c(1,2))

> hist(islands,breaks=”Sturges”,main=”Sturges”)

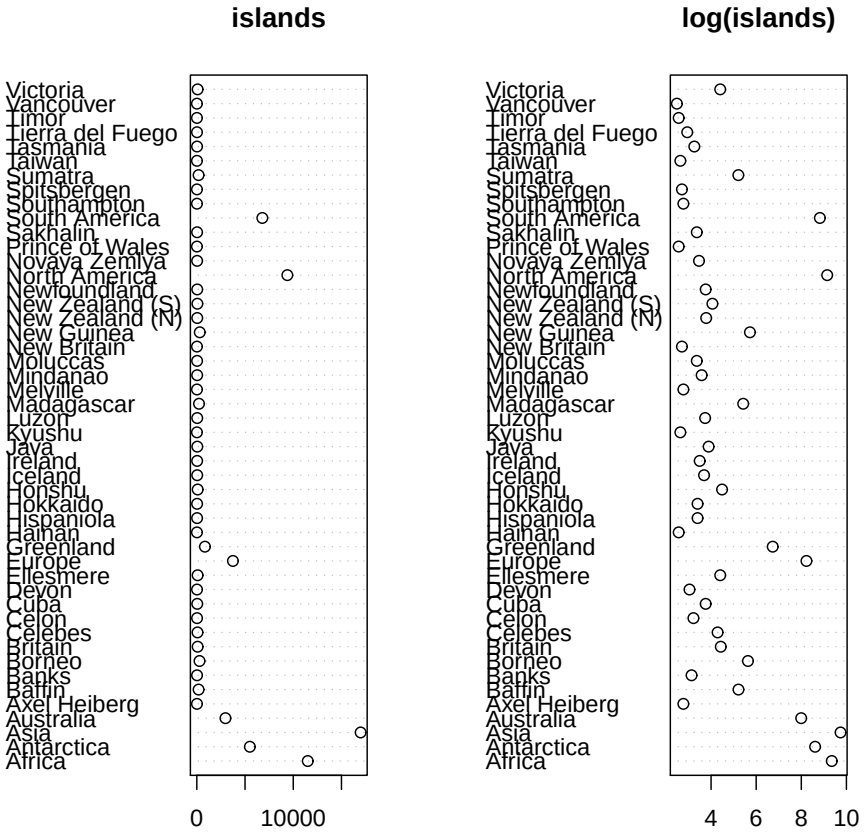
> hist(islands,breaks=”Scott”,main=”Scott”)



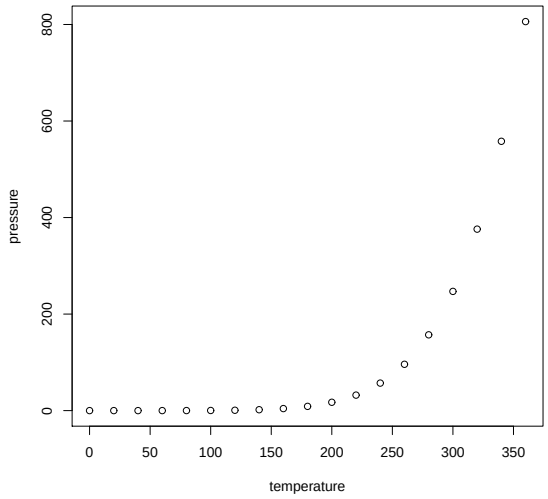

```
> par(mfrow=c(1,2))
> hist(log(islands),breaks="Sturges",main="Sturges")
> hist(log(islands),breaks="Scott",main="Scott")
```



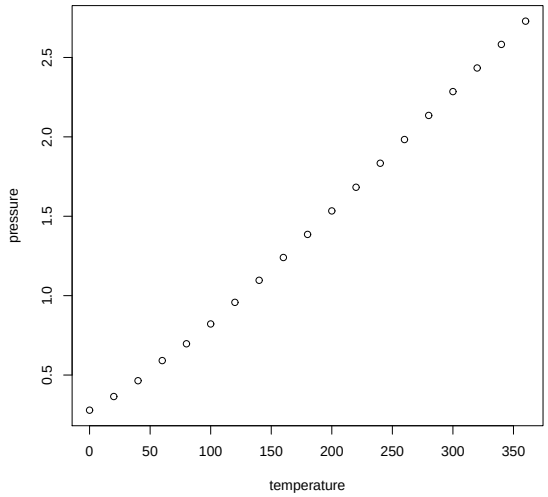
(e)
```
> par(mfrow=c(1,2))
> dotchart(islands,main="islands")
> dotchart(log(islands),main="log(islands)")
```



3. (a) >plot(pressure$temperature,pressure$pressure,xlab=”temperature”,ylab=”pressure”)

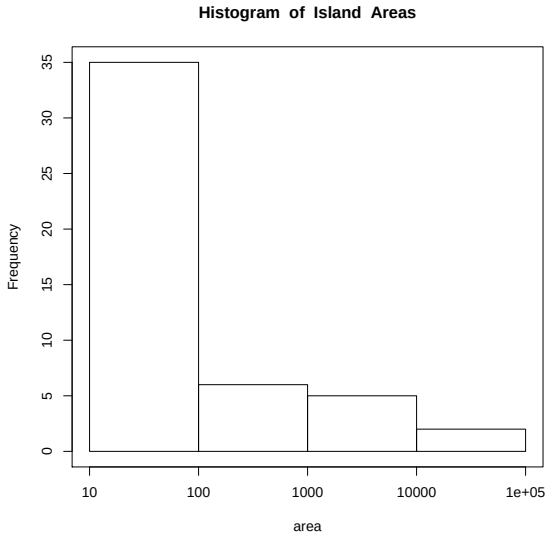


(c) >plot(pressure$temperature,pressure$pressure^(3/20),xlab=”temperature”,ylab=”pressure”)

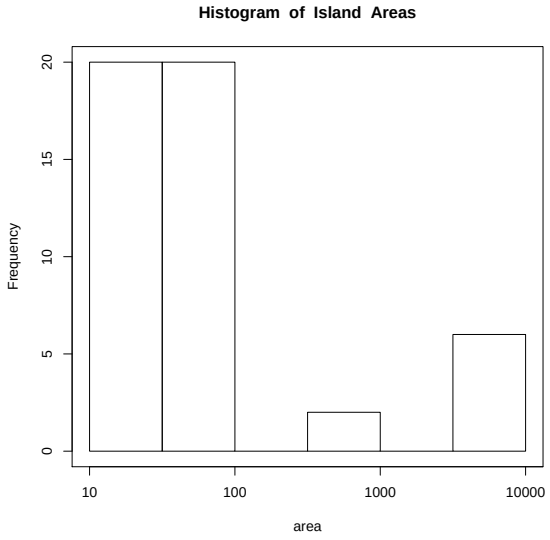


第三章习题

1. (a) > hist(log(islands,10),breaks="Scott",axes=FALSE,xlab="area",main="Histogram of Island Areas")
   > axis(1,at=1:5,labels=10^(1:5))
   > axis(2)
   > box()



(c) > hist(round(log(islands,10)),breaks="Sturges",axes=FALSE,xlab="area",
+ main="Histogram of Island Areas")
> axis(1,at=1:5,labels=10^(1:5))
> axis(2)
> box()



第四章

4.1

4.1.1

1. (a) > Fibonacci <- numeric(12)
   > Fibonacci[1] <- 2
   > Fibonacci[2] <- 2
   > for (i in 3:12) Fibonacci[i] <- Fibonacci[i-2] + Fibonacci[i-1]
   > Fibonacci
   [1] 2 2 4 6 10 16 26 42 68 110 178 288

   (b) > Fibonacci <- numeric(12)
   > Fibonacci[1] <- 3
   > Fibonacci[2] <- 2
   > for (i in 3:12) Fibonacci[i] <- Fibonacci[i-2] + Fibonacci[i-1]
   > Fibonacci
   [1] 3 2 5 7 12 19 31 50 81 131 212 343

   (c) > Fibonacci <- numeric(12)
   > Fibonacci[1] <- Fibonacci[2] <- 1
   > for (i in 3:12) Fibonacci[i] <- Fibonacci[i-1] - Fibonacci[i-2]
   > Fibonacci
   [1] 1 1 0 -1 -1 0 1 1 0 -1 -1 0

   (d) > Fibonacci <- numeric(12)

```
> Fibonacci[1] <- Fibonacci[2] <- Fibonacci[3] <- 1
> for (i in 4:12) Fibonacci[i] <- Fibonacci[i-1] + Fibonacci[i-2] + Fibonacci[i-3]
> Fibonacci
[1]  1  1  1  3  5  9  17  31  57  105  193  355
```

3. (a)
```
> answer <- 0
> for (j in 1:5) answer <- answer + j
> answer
[1] 15
```

(b)
```
> answer <- NULL
> for (j in 1:5) answer <- answer +j
> answer
numeric(0)
```

(c)
```
> answer <- 0
> for (j in 1:5) answer <- c(answer, j)
> answer
[1]  0  1  2  3  4  5
```

(d)
```
> answer <- 1
> for (j in 1:5) answer <- answer*j
> answer
[1] 120
```

(e)
```
> answer <- 3
> for (j in 1:15) answer <- c(answer, (7*answer[j])%%31)
> answer
[1]  3  21  23  6  11  15  12  22  30  24  13  29  17  26  27  3
```

5.
```
> x <- 0.5
> count <- 0
> while(abs(x-cos(x))>0.01){
+         x <- cos(x)
+         count <- count+1
+ }
> count
[1] 10
> x <- 0.5
```

```
> count <- 0
> while(abs(x-cos(x))>0.001){
+         x <- cos(x)
+         count <- count+1
+ }
> count
[1] 15
> x <- 0.5
> count <- 0
> while(abs(x-cos(x))>0.0001){
+         x <- cos(x)
+         count <- count+1
+ }
> count
[1] 21
> x <- 0.7
> count <- 0
> while(abs(x-cos(x))>0.0001){
+         x <- cos(x)
+         count <- count+1
+ }
> count
[1] 17
> x <- 0.0
> count <- 0
> while(abs(x-cos(x))>0.0001){
+         x <- cos(x)
+         count <- count+1
+ }
> count
[1] 23
```

4.1.2

1. 对，当 n 不是整数时，函数仍能正常工作。

3. ```
   > GIC <- function(P, n){
   +       if (n<=3) i <- 0.04 else i <- 0.05
   +       return(P*((1+i)^n-1))
   +   }
   ```

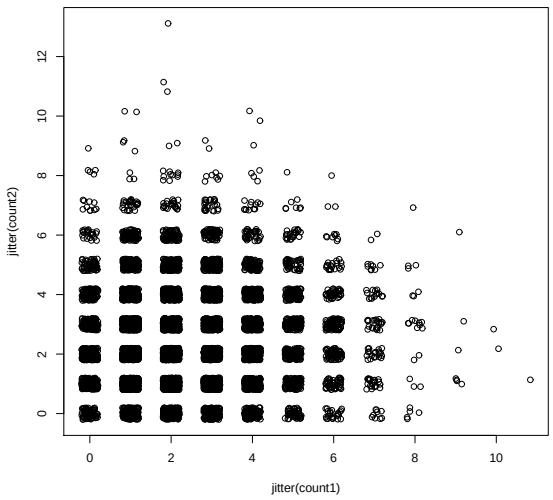
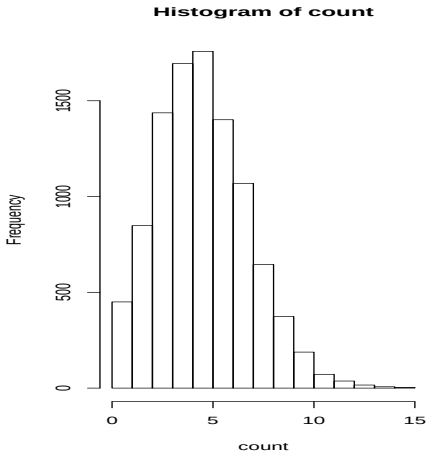
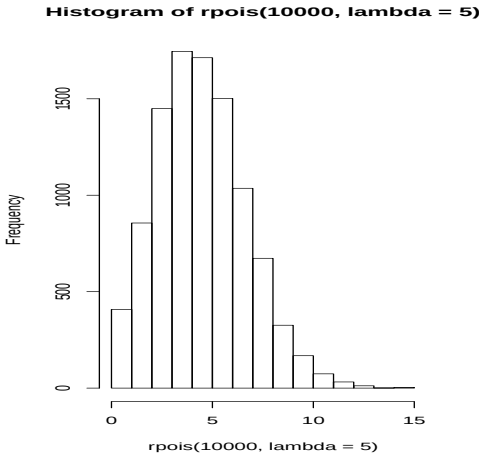
4.1.3

1. ```
   > Fib1 <- 1
   > Fib2 <- 1
   > Fibonacci <- c(Fib1, Fib2)
   > while (Fib1 < 300){
   +       Fibonacci <- c(Fibonacci, Fib2)
   +       Fib2 <- Fib1 + Fib2
   +       Fib1 <- max(Fibonacci)
   +  }
   > print(Fibonacci)
   [1]  1  1  1  2  3  5  8 13 21 34 55 89 144 233 377
   ```

3. ```
   > i <- 0
   > i0 <- 0.006
   > while  (abs(i  -  i0)>=  0.000001){
   +        i  <-  i0
   +        i0  <-  (1-(1+i)^(-20))/19
   +  }
   > print(i)
   [1]  0.004955135
   ```

5. ```
   > Fibonacci <- c(1, 1, 1)
   > while  (max(Fibonacci)<1000000){
   +       Fibonacci <- c(Fibonacci, Fibonacci[length(Fibonacci)]+
   +       Fibonacci[length(Fibonacci)-1])
   + }
   > print(length(Fibonacci))
   [1]  32
   ```

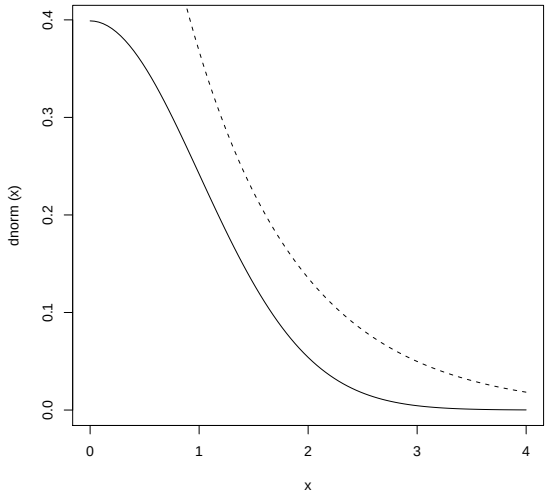
4.1.4

1. ```
   > x <- 0
   ```
   ```

```
> f <- x^7+10000*x^6+1.06*x^5+10600*x^4+0.0605*x^3+605*x^2+0.0005*x+5
> tolerance <- 0.000001
> count <- 0
> while (abs(f) > tolerance){
+  f.prime <-7*x^6+6*10000*x^5+5*1.06*x^4+4*10600*x^3+3*0.0605*x^2+2*605*x+0.0005
+  x <- x - f/f.prime
+  f <-x^7+10000*x^6+1.06*x^5+10600*x^4+0.0605*x^3+605*x^2+0.0005*x+5
+  count <- count+1
+ }
> count
[1] 1
```

3.
```
> x <- -1.5
> f <- cos(x) + exp(x)
> tolerance <- 0.000001
> while (abs(f) > tolerance){
+       f.prime <- -sin(x) - 1/x
+       x <- x - f/f.prime
+       f <- cos(x) + exp(x)
+ }
> x
[1] -1.746139
```

5. 函数只是在 $x = 0.6$ 时取得零值。对于初始估计值 0.5 , 0.75 和 0.2 来说，牛顿法给出的解是 $x = 1$ 。假如初始估计值是 1.25 ，那么用牛顿法就不收敛。

7.
```
> i <- 0.006
> count <- 0
> f <- (1- (1+i)^(-20))/19 - i
> tolerance <- 0.000001
> while (abs(f) > tolerance){
+       f.prime <- (20/19)*((1+i)^(-21))-1
+       i <- i - f/f.prime
+       f <- (1- (1+i)^(-20))/19 - i
+       count <- count+1
+ }
> i
```

[1] 0.004939979

> count

[1] 2

4.1.5

1. > x1 <- 0

> x2 <- 2

> repeat{

+ f1 <- x1^3+2*x1^2-7

+ f2 <- x2^3+2*x2^2-7

+ x3 <- (x1+x2)/2

+ f3 <- x3^3+2*x3^2-7

+ if(f3==0){

+ print(x3)

+ break

+ }else{

+ if(f3*f1>0){

+ x1 <- x3

+ }else{

+ x2 <- x3

+ }

+ }

+ if(abs(x1-x2)<0.000001){

+ print((x1+x2)/2)

+ break

+ }

+ }

[1] 1.428817

4.2

4.2.1

1. 输入函数名来调用函数，例如：

> var

function (x, y = NULL, na.rm = FALSE, use)

```
{
if (missing(use))
use <- if (na.rm)
"complete.obs"
else "all.obs"
na.method <- pmatch(use, c("all.obs", "complete.obs", "pairwise.complete.obs"))
if (is.data.frame(x))
x <- as.matrix(x)
else stopifnot(is.atomic(x))
if (is.data.frame(y))
y <- as.matrix(y)
else stopifnot(is.atomic(y))
.Internal(cov(x, y, na.method, FALSE))
}
<environment: namespace:stats>
```

3.
```
> bisection <- function(f, x1, x2){
+       repeat{
+ f1 <- f(x1)
+ f2 <- f(x2)
+ x3 <- (x1+x2)/2
+ f3 <- f(x3)
+ if(f3==0){
+ print(x3)
+ break
+ }else{
+ if(f3*f1>0){
+ x1 <- x3
+ }else{
+ x2 <- x3
+ }
+ }
+ if(abs(x1-x2)<0.000001){
+ print((x1+x2)/2)
```

+ break

+ }

+ }

+ }

4.3

1. > factorial(10)

   [1] 3628800

   > factorial(50)

   [1] 3.041409e+64

   > factorial(100)

   [1] 9.332622e+157

   > factorial(1000)

   [1] Inf

   Warning message:

   value out of range in 'gammafn'

3. > fix(compound.interest)

   > #change the function to:

   > function(P, j, m, n){

   + i.r <- j/m

   + return(P*(1+i.r)^(n*m))

   + }

   5. > mortgage.payment <- function(P, i.r, n){

   + R <- P*i.r/(1-(1+i.r)^(-n))

   + return(R)

   + }

4.4

4.4.1

1. > mergesort <- function(x, decreasing=FALSE){

   + if(decreasing==FALSE){

   + len <- length(x)

   + if (len<2) result <- x

   + else{

```
+ y <- x[1:(len/2)]
+ z <- x[((len/2)+1):len]
+ y <- mergesort(y)
+ z <- mergesort(z)
+ result <- c()
+ while(min(length(y), length(z))>0){
+ if(y[1]<z[1]){
+ result <- c(result, y[1])
+ y <- y[-1]
+ }else{
+ result <- c(result, z[1])
+ z <- z[-1]
+ }
+ }
+ if(length(y)>0)
+ result  <-  c(result,  y)
+ else
+ result  <-  c(result,  z)
+ }
+ return(result)
+       }else{
+ len <- length(x)
+ if (len<2) result <- x
+ else{
+ y <- x[1:(len/2)]
+ z <- x[((len/2)+1):len]
+ y <- mergesort(y, decreasing=TRUE)
+ z <- mergesort(z, decreasing=TRUE)
+ result <- c()
+ while(min(length(y),  length(z))>0){
+ if(y[1]>z[1]){
+ result <- c(result,  y[1])
+ y <- y[-1]
+ }else{
```

```
+ result <- c(result, z[1])
+ z <- z[-1]
+ }
+ }
+ if(length(y)>0)
+ result <- c(result, y)
+ else
+ result <- c(result, z)
+ }
+ return(result)
+         }
+    }
```

4.7

1.
```
> directpoly <- function(x, coef){
+ n <- length(coef)
+ result <- 0
+ for(i in 1:n){
+ result <- result+coef[i]*x^(n-i)
+ }
+ return(result)
+ }
```

2.
```
> hornerpoly <- function(x, coef){
+ n <- length(coef)
+ a <- matrix(0, length(x), n)
+ a[,n] <- coef[1]
+ for(i in (n-1):1){
+ a[,i] <- a[,i+1]*x+coef[n-i+1]
+ }
+ return(a[,1])
+ }
```

5. (a)
```
> x1 <- 2.1
> x2 <- 3.1
> count <- 1
```

```
> repeat{
+ count <- count+1
+ f1 <- (x1-3)*exp(-x1)
+ f2 <- (x2-3)*exp(-x2)
+ x3 <- (x1+x2)/2
+ f3 <- (x3-3)*exp(-x3)
+ if(f3==0){
+ break
+ }else{
+ if(f3*f1>0){
+ x1 <- x3
+ }else{
+ x2 <- x3
+ }
+ }
+ if(abs(x1-x2)<0.0000001){
+ break
+ }
+ }
> count
[1] 25
```

(b)

```
> x1 <- 2.1
> x2 <- 3.1
> count <- 1
> repeat{
+         count <- count+1
+         f1 <- (x1^2-6*x1+9)*exp(-x1)
+         f2 <- (x2^2-6*x2+9)*exp(-x2)
+         x3 <- (x1+x2)/2
+         f3 <- (x3^2-6*x3+9)*exp(-x3)
+         if(f3==0){
+ break
+         }else{
+ if(f3*f1>0){
```

```
+ x1 <- x3
+ }else{
+ x2 <- x3
+ }
+         }
+         if(abs(x1-x2)<0.0000001){
+ break
+         }
+  }
```

第五章

5.2

1.
```
> x0 <- 17218
> x <- numeric(20)
> x[1] <- (172*x0)%%30307
> for(i in 2:20){
+   x[i] <- (x[i-1]*172)%%30307
+   }
> x
 [1] 21717 7563 27942 17518 12703 2812 29059 27800 23401 24448 22690 23384
[13] 21524 4674 15946 15082 18009 6234 11503 8561
```

3. (a)
```
> set.seed(32078)
> runif(20,0,1)
 [1] 0.25646258 0.49881767 0.52665491 0.62698163 0.80527541 0.18434520
 [7] 0.51023267 0.36839051 0.17081759 0.74328879 0.01424726 0.22347861
[13] 0.36032442 0.79655794 0.01565888 0.40551753 0.65151347 0.91013292
[19] 0.77073083 0.51094885
```

(b)
```
> set.seed(32078)
> runif(20,3,7)
 [1] 4.025850 4.995271 5.106620 5.507927 6.221102 3.737381 5.040931 4.473562
 [9] 3.683270 5.973155 3.056989 3.893914 4.441298 6.186232 3.062636 4.622070
[17] 5.606054 6.640532 6.082923 5.043795
```

(c)
```
> set.seed(32078)
```

```
> runif(20,-2,2)
 [1] -0.974149697 -0.004729333  0.106619657  0.507926506  1.221101642
 [6] -1.262619189  0.040930690 -0.526437979 -1.316729628  0.973155177
[11] -1.943010969 -1.106085563 -0.558702309  1.186231747 -1.937364492
[16] -0.377929887  0.606053870  1.640531669  1.082923308  0.043795402
```

5. (a)
```
> r <- runif(10000, 3.7, 5.8)
> mean(r)
[1] 4.756022
> var(r)
[1] 0.3664445
> sd(r)
[1] 0.6053466
```

(b)
```
> length(r[r>4])/length(r)
[1] 0.8584
```

7.
```
> U1 <- runif(10000)
> U2 <- runif(10000)
> U3 <- runif(10000)
> U <- U1+U2+U3
```

(a)
```
> mean(U)
[1] 1.50653
```

(b)
```
> var(U)
[1] 0.2548007
> var(U1)+var(U2)+var(U3)
[1] 0.251535
```

(c)
```
>  mean(sqrt(U))
[1]  1.207929
```

(d)
```
> V <- sqrt(U1)+sqrt(U2)+sqrt(U3)
> length(V[V>=.8])/length(V)
[1]  0.9965
```

9. (a)
```
> sample(1:100, size=50, replace=F)
 [1] 26 71 83 86 60 37 85 97 34 33 25 91 78 15 74 69 84 82 63 64 35 40 73 38 16
[26] 32 22 43 19 88 68 12 8  80 17 81 99 4  66 79 49 11 31  1 61  7 46 27 39  9
```

(b) > sample(1:100, size=50, replace=T)

[1] 80 32 67 86 77 87 48 41 85 53 29 17 76 72 66 56 32 36 17 61 27 87 8 2 18

[26] 42 88 99 60 80 38 25 68 79 98 9 40 34 16 72 18 87 15 65 8 65 3 90 15 64

5.3

5.3.1

1. (a) > r <- rbinom(n=10, size=1, p=.5)

> sum(r)

[1] 6

(b) > r <- rbinom(n=1000, size=1, p=.5)

> sum(r)

[1] 481

3. > r <- rbinom(n=500, size=1, p=.99)

> mean(r)

[1] 0.99

> var(r)

[1] 0.00991984

理论值是 .99 和 .0099.

5.3.2

1. > rbinom(p=0.15, size=25, n=24)

[1] 5 4 5 5 4 2 3 3 1 2 4 2 3 2 5 5 4 5 3 3 3 8 5 4

> # Exceeds 5 once.

> rbinom(p=0.2, size=25, n=24)

[1] 5 6 5 5 4 8 8 4 5 7 7 5 7 3 6 4 5 4 4 6 6 6 7 9

> # Exceeds 5 several times.

> rbinom(p=0.25, size=25, n=24)

[1] 6 3 1 13 5 11 11 3 4 7 9 4 4 1 5 6 6 7 9 6 9 7 7 9

3. > r <- rbinom(n=1000, size=18, p=.76)

> mean(r)

[1] 13.697

> var(r)

[1] 3.472664

理论值是 13.68 和 3.2832.

5. (a)
```
> #Generate binomial pseudorandom variables using inversion method
> ranbin2 <- function(n, size, prob){
+ # 'singlenumber' generates one binomial random variable.
+ singlenumber <- function(size, prob){
+ x <- runif(size)
+ N <- sum(x<prob)
+ N
+         }
+     replicate(n, singlenumber(size, prob))
+  }
```

(b)
```
> system.time(gcFirst=T, ranbin(n=10000, size=10, prob=.4))
user   system  elapsed
0.15     0.00     0.16
> system.time(gcFirst=T, rbinom(n=10000, size=10, prob=.4))
user    system  elapsed
0.02      0.00     0.02
> system.time(gcFirst=T, ranbin(n=10000, size=100, prob=.4))
user    system  elapsed
0.18      0.00     0.19
> system.time(gcFirst=T, rbinom(n=10000, size=100,  prob=.4))
user    system  elapsed
  0         0        0
> system.time(gcFirst=T, ranbin(n=10000, size=1000, prob=.4))
user    system  elapsed
0.33      0.00     0.33
> system.time(gcFirst=T, rbinom(n=10000, size=1000, prob=.4))
user    system  elapsed
0.01      0.00     0.02
```

5.3.3

1.
```
> rpois(n=15, lambda=2.8)
[1] 1 2 4 3 3 4 2 5 4 2 4 1 2 2 1
```

3.
```
> x <- rpois(10000, lambda=7.2)
> mean(x)
```

[1] 7.141

> var(x)

[1] 7.287248

7. > N <- rpois(10000, lambda=2.5 * 2)

   (a) > count1 <- c()
   > count2 <- c()
   > for(i in 1:10000){
   + u <- runif(N[i], max=2)
   + count1[i] <- sum(u<1)
   + count2[i] <- sum(u>1)
   + }
   > sum(count1)
   [1] 24886
   > sum(count2)
   [1] 24889

   (b) yes

   (c) > plot(jitter(count1), jitter(count2))



5.3.4

1. > r <- rexp(50000, rate=3)

   (a) > sum(r<1)/length(r)

```
[1] 0.95204
> pexp(1, rate=3)
[1] 0.950213
```

(b)
```
> mean(r)
[1] 0.3309469
```

(c)
```
> var(r)
[1] 0.1083953
```

3.
```
> r1 <- rexp(100000, rate=1/3)
> r2 <- rexp(100000, rate=1/6)
> index <- (r2-r1)>0
> r.min <- c(r1[index], r2[!index])
> mean(r.min)
[1] 1.984397
> var(r.min)
[1] 3.961419
```

5.
```
> count <- c()
> for(i in 1:10000){
+         cum <- 0
+         j <- 0
+         while(cum<=2){
+ j <- j+1
+ r <- rexp(1, rate=2.5)
+ cum <- cum + r
+         }
+         count[i] <- j-1
+  }
>  par(mfrow=c(1,2))
>  hist(count,  breaks=20)
>  hist(rpois(10000, lambda=5), breaks=20)
```





在这使用 QQ 图也是一个很好的选择。

5.3.5

1. > r <- rnorm(n=100, mean=51, sd=5.2)
   > mean(r)
   [1] 51.63488
   > sd(r)
   [1] 4.797961

3. > sim <- function(){
   + ans <- 0
   + while(abs(ans)<=2){
   + ans <- rnorm(1, mean=3, sd=4)
   + }
   + ans
   +}
   > sim()
   [1] 7.188694

5. > r <- rchisq(n=100, df=8)
   > mean(r)
   [1] 8.36974
   > var(r)
   [1] 18.08655

5.4

1. > #Monte Carlo

```
> u <- runif(1000)
> mean(u)
[1] 0.5140157
> #Integrate function
> f <- function(x){
+       x
+ }
> integrate(f, lower=0, upper=1)
0.5 with absolute error < 5.6e-15
> #Monte  Carlo
> u <- runif(1000,  max=3)
> mean(u^2)*(3-0)
[1] 9.138817
> #Integrate  function
> f <- function(x){
+       x^2
+ }
> integrate(f, lower=0, upper=3)
9 with absolute error < 1e-13
> #Monte Carlo
> u <- runif(1000, max=pi)
> mean(sin(u))*(pi-0)
[1] 1.988680
> #Integrate function
> f <- function(x){
+       sin(x)
+ }
> integrate(f, lower=0, upper=pi)
2 with absolute error < 2.2e-14
>  #Monte  Carlo
> u <- runif(1000, min=1, max=pi)
> mean(exp(u)/dunif(u,min=1, max=pi))
[1] 20.51446
> #Integrate  function
```

```
> f <- function(x){
+       exp(x)
+ }
> integrate(f, lower=1, upper=pi)
20.42241 with absolute error < 2.3e-13
> #Monte Carlo
> r <- rexp(1000)
> mean(exp(-r)/dexp(r))
[1] 1
> #Integrate  function
> f <- function(x){
+       exp(-x)
+ }
> integrate(f, lower=0, upper=1000)
1 with absolute error < 9e-10
> #Monte Carlo
> r <- rexp(1000)
> mean(exp(-r^3)/dexp(r))
[1] 0.9187494
> #Integrate  function
> f <- function(x){
+      exp(-x^3)
+ }
> integrate(f, lower=0, upper=1000)
0.8929795 with absolute error < 8.4e-10
> #Monte Carlo
> r <- runif(1000, max=3)
> mean(sin(exp(r))/dunif(r,  max=3))
[1] 0.617729
> #Integrate  function
> f <- function(x){
+        sin(exp(x))
+ }
> integrate(f,  lower=0,  upper=3)
```

```
0.6061245 with absolute error < 2.8e-10
> #Monte Carlo
> r <- runif(1000,max=1)
> mean(dnorm(r))
[1] 0.3382709
> #Integrate function
> f <- function(x){
+       dnorm(x)
+ }
> integrate(f, lower=0, upper=1)
0.3413447 with absolute error < 3.8e-15
> #Monte Carlo
> r <- runif(1000,max=2)
> mean(dnorm(r)*2)
[1] 0.4830885
> #Integrate function
> f <- function(x){
+       dnorm(x)
+ }
> integrate(f, lower=0, upper=2)
0.4772499 with absolute error < 5.3e-15
> #Integrate function
> f <- function(x){
+       dnorm(x)
+ }
> integrate(f, lower=0, upper=3)
0.4986501 with absolute error < 5.5e-15
```

5.5

5.5.2

1. ```
   > rand.normal <- function(n){
   +     r1 <- runif(n, min=-4, max=4)
   +     r2 <- runif(n)
   +     ans <- r1[ r2 < 2.5*dnorm(r1)]
   ```
   ```

```
+     while(length(ans) < n){
+ r1 <- runif(n, min=-4, max=4)
+ r2 <- runif(n)
+ ans <- c(ans,r1[ r2 < 2.5*dnorm(r1)])
+     }
+     ans[1:n]
+ }
```

3. > r <- rannorm(10000,8,2)

5. > curve(dnorm, from =0, to =4)
   > curve(dexp, from =0, to =4, add=T, lty=2)



7. > set.seed(91626)
   > probs <- runif(100)
   > probs <- probs/sum(probs)
   > system.time(gcFirst=T, r <- randiscrete1(100, probs))
   user system elapsed
   0   0   0
   > system.time(gcFirst=T, r <- randiscrete1(1000, probs))
   user system elapsed
   0.02   0.00   0.02
   > system.time(gcFirst=T, r <- randiscrete1(10000, probs))
   user system elapsed

```
0.2    0.0    0.2
> system.time(gcFirst=T, r <- randiscrete2(100, probs))
user  system elapsed
  0     0     0
> system.time(gcFirst=T, r <- randiscrete2(1000, probs))
user  system elapsed
0.06    0.00    0.06
> system.time(gcFirst=T, r <- randiscrete2(10000, probs))
user  system elapsed
0.69    0.00    0.69
```

When inversion method is not available.

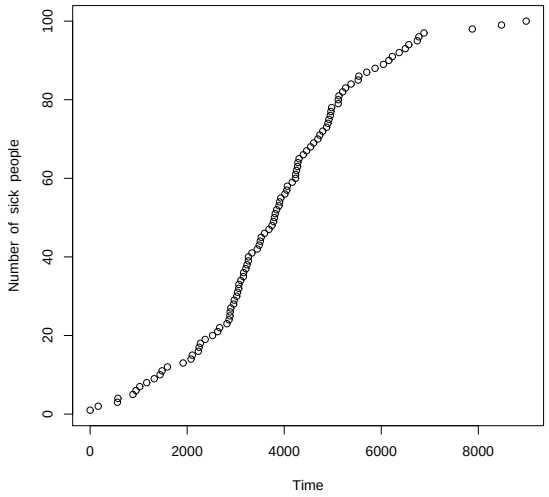
5.6

1.
```
> simulate <- function(AnnFirst = T, pAnn, pBob){
+     scoreAnn <- 0
+     scoreBob <- 0
+     nextPlayer <- ' Ann '
+     if(AnnFirst==F){
+ nextPlayer <- Bob
+     }
+     while(scoreAnn < 21 & scoreBob < 21){
+ if(nextPlayer== ' Ann ' ){
+ r <- rbinom(1, 1, pAnn)
+ if(r==1){
+ scoreAnn <- scoreAnn + 1
+ }else{
+ nextPlayer <- ' Bob '
+ }
+ }else{
+ r <- rbinom(1, 1, pBob)
+ if(r==1){
+ scoreBob <- scoreBob + 1
+ }else{
+ nextPlayer <- ' Ann '
```

```
+ }
+ }
+     }
+     return(nextPlayer)
+ }
> pAnn <- .7
> pBob <- .72
> result <- c()
> for(i in 1:7){
+     result <- c(result,simulate(AnnFirst=T, pAnn=pAnn, pBob=pBob))
+ }
```

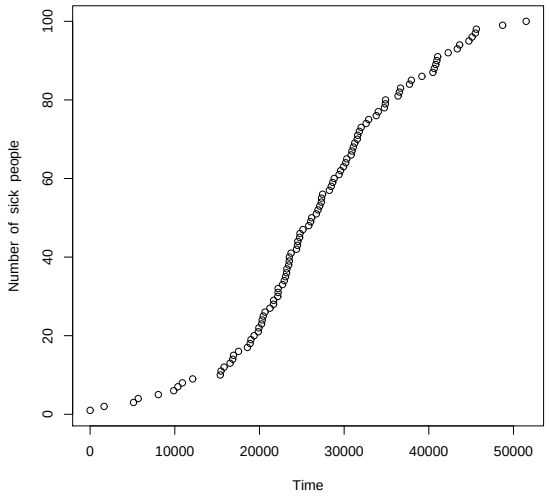
2.
```
> simulate <- function(N, p){
+     time.line <- 0
+     time.current <- 0
+     n.sick <- 1
+     while(n.sick<N){
+ time.current <- time.current + 1
+
+ #if encounters==0, then none of the two persons is sick
+ #if encounters==1, then one of the two persons is sick
+ #if encounters==2, then both of the two persons are sick
+ encounters <- rhyper(nn=1, m=n.sick, n=N-n.sick, k=2)
+
+ if(encounters==1){
+ contage <- rbinom(n=1, size=1, p)
+ if(contage==1){
+ #record current time
+ time.line <- c(time.line, time.current)
+ n.sick <- n.sick + 1
+ }
+ }
+     }
+     time.line
+ }
```

```
> N <- 100
> p <- .05
> ans <- simulate(N, p)
> plot(ans, 1:N, xlab= 'Time', ylab= 'Number of sick people' )
```



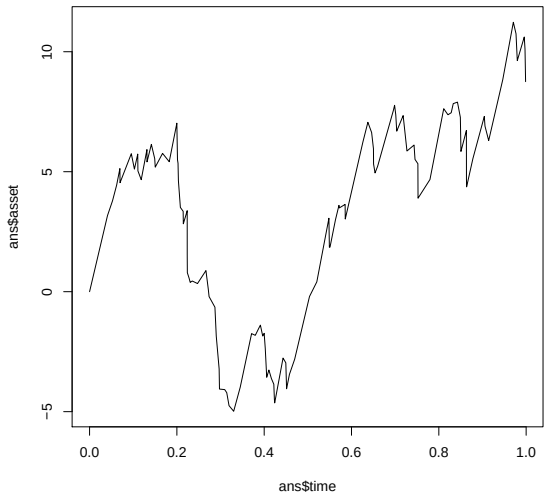
5.
```
> simulate <- function(N, p){
+     time.line <- 0
+     time.current <- 0
+     n.sick <- 1
+
+     while(n.sick<N){
+ time.current <- time.current + rexp(n=1,rate=1/5)
+
+ #if encounters==0, then none of the two persons is sick
+ #if encounters==1, then one of the two persons is sick
+ #if encounters==2, then both of the two persons are sick
+ encounters <- rhyper(nn=1, m=n.sick, n=N-n.sick, k=2)
+
+ if(encounters==1){
+ contage <- rbinom(n=1, size=1, p)
+ if(contage==1){
+ #record current time
+ time.line <- c(time.line, time.current)
```

```
+ n.sick <- n.sick + 1
+ }
+ }
+     }
+     time.line
+ }
> N <- 100
> p <- .05
> ans <- simulate(N, p)
> plot(ans, 1:N, xlab= ' Time ' , ylab= ' Number of sick people ' )
```



7. (a)
```
> simulate <- function(){
+     N <- rpois(n=1, lambda=100)
+     claimSize <- rgamma(N, shape=2, rate=2)
+     claimTime <- sort(runif(N))
+
+     asset <- 0
+     asset.1 <- 105 * claimTime[1] - claimSize[1]
+     asset <- c(asset, asset.1)
+
+     for(i in 2:N){
+ len <- length(asset)
+ asset.i <- asset[len] + (claimTime[i] - claimTime[i-1])*105 - claimSize[i]
```

```
+ asset <- c(asset, asset.i)
+     }
+     claimTime <- c(0, claimTime)
+     list(time=claimTime, asset=asset)
+ }
> ans <- simulate()
> plot(ans$time, ans$asset, type= ' l ' )
```



(b)
```
> minimum <- c()
> final <- c()
> for(i in 1:1000){
+     ans <- simulate()
+     minimum <- c(minimum, min(ans$asset))
+     final <- c(final, rev(ans$asset)[1])
+ }
> mean(minimum)
[1] -6.723402
> mean(final)
[1] 4.721359
```

第六章

6.1

6.1.1

1. >A<-matrix(rep(seq(1,5),5),nrow=5)+matrix(rep(seq(0,4),5),byrow=TRUE,nrow=5)

   >A

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	2	3	4	5
[2,]	2	3	4	5	6
[3,]	3	4	5	6	7
[4,]	4	5	6	7	8
[5,]	5	6	7	8	9

   > Hankel <- function(x){

   + A<-matrix(rep(seq(1,x),x),nrow=x)+matrix(rep(seq(0,x-1),x),byrow=TRUE,nrow=x)

   + return(A)

   + }

   > Hankel(10)

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]	[,9]	[,10]
[1,]	1	2	3	4	5	6	7	8	9	10
[2,]	2	3	4	5	6	7	8	9	10	11
[3,]	3	4	5	6	7	8	9	10	11	12
[4,]	4	5	6	7	8	9	10	11	12	13
[5,]	5	6	7	8	9	10	11	12	13	14
[6,]	6	7	8	9	10	11	12	13	14	15
[7,]	7	8	9	10	11	12	13	14	15	16
[8,]	8	9	10	11	12	13	14	15	16	17
[9,]	9	10	11	12	13	14	15	16	17	18
[10,]	10	11	12	13	14	15	16	17	18	19

   > Hankel(12)

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]	[,9]	[,10]	[,11]	[,12]
[1,]	1	2	3	4	5	6	7	8	9	10	11	12
[2,]	2	3	4	5	6	7	8	9	10	11	12	13
[3,]	3	4	5	6	7	8	9	10	11	12	13	14
[4,]	4	5	6	7	8	9	10	11	12	13	14	15
[5,]	5	6	7	8	9	10	11	12	13	14	15	16
[6,]	6	7	8	9	10	11	12	13	14	15	16	17
[7,]	7	8	9	10	11	12	13	14	15	16	17	18
[8,]	8	9	10	11	12	13	14	15	16	17	18	19
[9,]	9	10	11	12	13	14	15	16	17	18	19	20
[10,]	10	11	12	13	14	15	16	17	18	19	20	21
[11,]	11	12	13	14	15	16	17	18	19	20	21	22
[12,]	12	13	14	15	16	17	18	19	20	21	22	23

3. > W <- cbind(c(1,1,1,1,1,1,1), c(2,3,4,5,6,7,8), c(4,7,5,6,7,5,3))

   > W

	[, 1]	[, 2]	[, 3]
[1,]	1	2	4
[2,]	1	3	7
[3,]	1	4	5
[4,]	1	5	6
[5,]	1	6	7
[6,]	1	7	5
[7,]	1	8	3

6.1.2 ACCESSING MATRIX ELEMENTS; ROW AND COLUMN NAMES

1. > Stochastic_matrix <- matrix(c(0.2, 0.3, 0.8, 0.7), ncol=2)

   > rownames(Stochastic_matrix) <- c("sunny", "rainy")

   > colnames(Stochastic_matrix) <- c("sunny", "rainy")

   > Stochastic_matrix

	sunny	rainy
sunny	0.2	0.8
rainy	0.3	0.7

3. > matrix[3, 1] <- 162

   > matrix[5, 1] <- 181

   > matrix[5, 2] <- 68

   > matrix

	height	weight
Neil	172	62
Cindy	168	64
Pardeep	162	51
Deepak	175	71
Hao	181	68

6.1.3

1. > A <- matrix(c(3, 5, 4, 8), ncol=2)

   > det(A)

   [1] 4

   > det(t(A))

   [1] 4

   > A <- matrix(c(3, 20, 15, 7), ncol=2)

   > det(A)

   [1] -279

   > det(t(A))

   [1] -279

   > A <- matrix(c(4, 2, 25, 23), ncol=2)

   > det(A)

[1] 42

> det(t(A))

[1] 42

6.1.4 TRIANGULAR MATRICES

1. > H3<- matrix(c(1,1/2,1/3,1/2,1/3,1/4,1/3,1/4,1/5),nrow=3)

   > Hnew <- H3

   > Hnew[lower.tri(H3, diag=TRUE)] <- 0

   > Hnew

	[,1]	[,2]	[,3]
[1,]	0	0.5	0.3333333
[2,]	0	0.0	0.2500000
[3,]	0	0.0	0.0000000

3. > X <- matrix(c(1, 2, 3, 1, 4, 9), ncol=2)

   > X[3, 2]

   [1] 9

   > X[3, 2, drop=FALSE]

   [,1]

   [1,] 9

   > dim(X[3, 2])

   NULL

   > dim(X[3, 2, drop=FALSE])

   [1] 1 1

6.2

1. > X <- matrix(c(1, 2, 3, 1, 4, 9), ncol=2)

   > 1.5*X

	[,1]	[,2]
[1,]	1.5	1.5
[2,]	3.0	6.0
[3,]	4.5	13.5

6.2.3

1. > XX <- t(X) %*% X

   > XXinv <- solve(XX)

   > XXinv

	[,1]	[,2]
[1,]	1.2894737	-0.4736842
[2,]	-0.4736842	0.1842105

```
> # Verification:
> crossprod(XX, XXinv)

              [,1]          [,2]
[1,]  1.000000e+00 -1.332268e-15
[2,] -1.887379e-15  1.000000e+00
```

数值上非常接近单位矩阵。

3. (a)
```
> Hilbert <- function(n) {
+     A <- matrix(rep(NA, n*n), nrow=n)
+     for(i in 1:n){
+ for(j in 1:n){
+ A[i, j] <- 1/(i+j-1)
+ }
+     }
+ return(A)
+ }
> # or
>
> Hilbert <- function(n) {
+     outer(1:n, 1:n, function(x, y) 1/(x+y-1))
+ }
```

(b) 对。（说明并非所有的希尔伯特矩阵都可逆，可参阅http://planetmath.org/encyclopedia/HilbertMatrix.html）

(c)
```
> qr.solve(Hilbert(1))
     [,1]
[1,]    1
```

下面给出 qr.solve(Hilbert(2)) 到 qr.solve(Hilbert(6)) 的结果。

```
     [,1] [,2]
[1,]    4   -6
[2,]   -6   12

     [,1] [,2] [,3]
[1,]    9  -36   30
[2,]  -36  192 -180
[3,]   30 -180  180

     [,1] [,2] [,3] [,4]
[1,]   16 -120  240 -140
[2,] -120 1200 -2700 1680
[3,]  240 -2700 6480 -4200
[4,] -140 1680 -4200 2800
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	25	-300	1050	-1400	630
[2,]	-300	4800	-18900	26880	-12600
[3,]	1050	-18900	79380	-117600	56700
[4,]	-1400	26880	-117600	179200	-88200
[5,]	630	-12600	56700	-88200	44100

[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	
[1,]	36	-630	3360	-7560	7560	-2772
[2,]	-630	14700	-88200	211680	-220500	83160
[3,]	3360	-88200	564480	-1411200	1512000	-582120
[4,]	-7560	211680	-1411200	3628800	-3969000	1552320
[5,]	7560	-220500	1512000	-3969000	4410000	-1746360
[6,]	-2772	83160	-582120	1552320	-1746360	698544

(d) > eigen(Hilbert(10))$values

[1] 1.751920e+00 3.429295e-01 3.574182e-02 2.530891e-03 1.287496e-04

[6] 4.729689e-06 1.228968e-07 2.147439e-09 2.266746e-11 1.093249e-13

其中一个特征值接近 $10e-13$，这是一个非常小的数字，而最大的特征值却又大于 1，所以矩阵的列之间存在线性相关。

6.2.4

1. > X1 <- c(10, 11, 12, 13, 14, 15)

> X2 <- X1^2

> X3 <- X1^3

> X4 <- X1^4

> X5 <- X1^5

> X0 <- c(1, 1, 1, 1, 1, 1)

> A <- cbind(X0, X1, X2, X3, X4, X5)

> f <- matrix(c(25, 16, 26, 19, 21, 20), nrow=6)

> a <- solve(A, f)

> a

	[,1]
X0	2.536100e+05
X1	-1.025510e+05
X2	1.650092e+04
X3	-1.320667e+03
X4	5.258333e+01
X5	-8.333333e-01

> A %*% a

	[,1]
[1,]	25
[2,]	16
[3,]	26
[4,]	19
[5,]	21
[6,]	20

> x <- c(10, 11, 12, 13, 14, 15)

> y <- c(25, 16, 26, 19, 21, 20)

> A <- outer(x, 0:5, function(x, y) x^y)

> A

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]
[1,]	1	10	100	1000	10000	100000
[2,]	1	11	121	1331	14641	161051
[3,]	1	12	144	1728	20736	248832
[4,]	1	13	169	2197	28561	371293
[5,]	1	14	196	2744	38416	537824
[6,]	1	15	225	3375	50625	759375

> A <- outer(x, 0:5, function(x, y) x^y)

> solve(A, y) # Solve Aa = y for a

[1] 2.536100e+05 -1.025510e+05 1.650092e+04 -1.320667e+03 5.258333e+01

[6] -8.333333e-01

$f(x) = 253610 - 102551x + 16500.92x^2 - 1320.667x^3 + 52.58333x^4 - .8333x^5$

6.3

1. > X <- matrix(c(1, 2, 3, 1, 4, 9), ncol=2)

   > H <- X %*% solve(t(X) %*% X) %*% t(X)

   > H

	[,1]	[,2]	[,3]
[1,]	0.5263158	0.4736842	-0.1578947
[2,]	0.4736842	0.5263158	0.1578947
[3,]	-0.1578947	0.1578947	0.9473684

3. > sum(E$values)

   [1] 2

   > trace <- function(data)sum(diag(data))

   > trace(H)

   [1] 2

5. Hint: Note that HX = X and H(I - X) = 0.

6.5

1. (a) > P <- matrix(c(0.1, 0.4, 0.3, 0.2,

   + 0.2, 0.1, 0.4, 0.3,

   + 0.3, 0.2, 0.1, 0.4,

   + 0.4, 0.3, 0.2, 0.1), nrow=4)

```
> apply(P,  1,  sum)
[1]  1  1  1  1
```

(b)
```
> P2 <- P %*% P
> P3 <- P %*% P %*% P
> P5 <- P2 %*% P3
> P10 <- P5 %*% P5
> P10
             [,1]         [,2]         [,3]         [,4]
[1,]  0.2500000  0.2500016  0.2500000  0.2499983
[2,]  0.2499983  0.2500000  0.2500016  0.2500000
[3,]  0.2500000  0.2499983  0.2500000  0.2500016
[4,]  0.2500016  0.2500000  0.2499983  0.2500000
```

(c)
```
> solve(rbind(rep(1,4),(diag(rep(1,4)) - t(P))[-4,]), c(1,0,0,0))
[1]  0.25  0.25  0.25  0.25
```
行与列的 P^n 都收敛于 x。

(d)
```
> set.seed(361)
> pseudo <- function(n) {
+     Y <- numeric(n)
+     Y[1] <- 1
+     for(j in 2:n) {
+ Y[j] <- sample(1:4, 1,
+ prob=P[Y[j-1], ],
+ replace=T)
+     }
+     return(Y)
+ }
> yresult <- pseudo(10000)
```

(e)
```
>  table(yresult)
yresult
1     2     3     4
2502  2508  2523  2467
```
相对频率分布接近 x。

3. (a)
```
> C <- matrix(c(245921, 7304620, 34390.48, 6864693), ncol=1)
> A <- matrix(c(1, 10, 0.1, 5,
+ 1, 30, 0.15, 27,
+ 1, 50, 0.03, 45,
```

```
+ 1, 100, 0.5, 125), nrow=4)
> X <- solve(A, C)
> X
          [,1]
[1,]  113750
[2,]   75324
[3,]   35546
[4,]   21301
```

(b)
```
> income <- c(X[1,1]*10,X[2,1]*30,X[3,1]*50,X[4,1]*100)
> income
[1]  1137500  2259720  1777300  2130100
>claimsize<-c(X[1,1]*0.1*50,X[2,1]*0.15*180,X[3,1]*0.03*1500,X[4,1]*0.5*250)
>claimsize
[1]  568750  2033748  1599570  2662625
```

(c)
```
> claimamount <- numeric(1000)
> for(i in 1:1000){
+ A <- rgamma(rpois(1, 0.1*X[1, 1]), shape=2, scale=50/2)
+ B <- rgamma(rpois(1, 0.15*X[2, 1]), shape=2, scale=180/2)
+ C <- rgamma(rpois(1, 0.03*X[3, 1]), shape=2, scale=1500/2)
+ D <- rgamma(rpois(1, 0.5*X[4, 1]), shape=2, scale=250/2)
+ claimamount[i] <- sum(A, B, C, D)
+ }
> mean(claimamount)
[1]  6863177
> var(claimamount)
[1] 4846302489
> sd(claimamount)
[1] 69615.39
> sum(claimamount > 7304620)/length(claimamount)
[1]  0
```

第七章

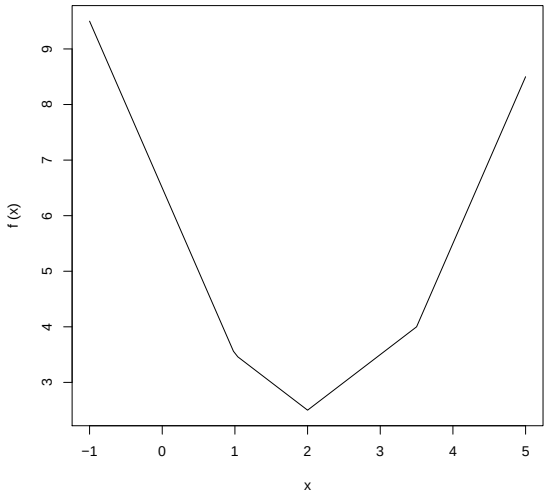
7.1

写出黄金分割搜索法函数

```
> golden <- function(f, a, b, tol = 0.0000001){
+    ratio <- 2/(sqrt(5) + 1)
```

```
+    x1 <- b - ratio*(b - a)
+    x2 <- a + ratio*(b - a)
+
+    f1 <- f(x1)
+    f2 <- f(x2)
+
+    while( abs(b -a) > tol){
+ if(f2 > f1){
+ b <- x2
+ x2 <- x1
+ f2 <- f1
+ x1 <- b - ratio*(b - a)
+ f1 <- f(x1)
+ }else{
+ a <- x1
+ x1 <- x2
+ f1 <- f2
+ x2 <- a + ratio*(b - a)
+ f2 <- f(x2)
+ }
+    }
+    return((a + b) / 2)
+ }
```

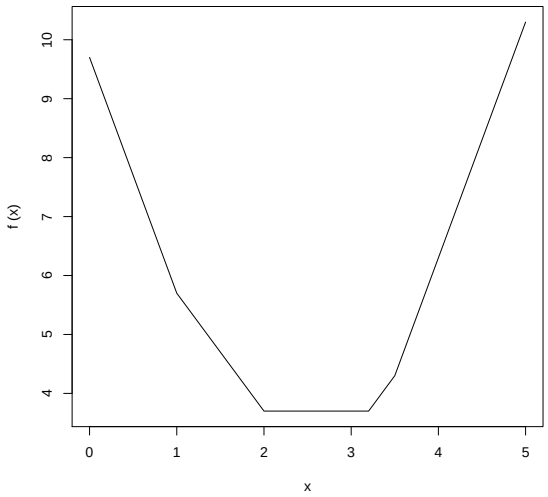
1. (a) `> f <- function(x) {`
   `+  abs(x - 3.5) + abs(x - 2) + abs(x - 1)`
   `+ }`
   `> curve ( f, from = -1, to = 5)`



这表明只有唯一接近于零的最小值，并且它位于零的右侧。

```
> golden( f, 1, 5)
[1] 2
```

(b)
```
> f <- function(x) {
+ abs(x - 3.2) + abs(x - 3.5) + abs(x - 2) + abs(x - 1)
+ }
> curve(f, from = 0, to = 5)
```



表明在 1 到 4 之间有多个最小值。

```
> golden( f, 1, 2.15)
[1] 2.15
```

```
> golden( f, 1, 5)
[1] 3.2
> golden( f, 1, 3)
[1] 3
> golden( f, 2.5, 3.5)
[1] 3.2
```

3.
```
> golden <- function( f, a, b, tot = 0.0000001){
+    ratio <- 2/(sqrt(5) + 1)
+    x1 <- b - ratio*(b - a)
+    x2 <- a + ratio*(b - a)
+
+    f1 <- f(x1)
+    f2 <- f(x2)
+
+    while( abs(b -a) > tot){
+ if(f2 < f1){
+ b <- x2
+ x2 <- x1
+ f2 <- f1
+ x1 <- b - ratio*(b - a)
+ f1 <- f(x1)
+ }else{
+ a <- x1
+ x1 <- x2
+ f1 <- f2
+ x2 <- a + ratio*(b - a)
+ f2 <- f(x2)
+ }
+    }
+    return((a + b)/2)
+ }
```

7.4

1. (a)
```
> f <- function(x) {
```

+ abs(x - 3.5) + abs(x - 2) + abs(x - 1)

+ }

> ans <- optimize(f, c(1, 3))

> ans$minimum

[1] 2

This confirms the minimum as 2.

(b) > f <- function(x) {

+ abs(x - 3.2) + abs(x - 3.5) + abs(x - 2) + abs(x - 1)

+ }

> ans <- optimize(f, c(1, 4))

> ans$minimum

[1] 2.145956
这再次确认最小值是 2.658372。

7.5

7.5.7

1. (a) > library(lpSolve)

> a.lp <- lp(objective.in = c(1, 3, 4, 1),

+ const.mat = matrix(c(1, -2, 0, 0,

+ 0, 3, 1, 0,

+ 0, 1, 0, 1), nrow = 3, byrow = TRUE),

+ const.rhs = c(9, 9, 10),

+ const.dir = c(">=", ">=", ">="))

> a.lp

Success: the objective function is 31

> a.lp$solution

[1] 15 3 0 7

(b) 结果不会改变，因为最优解要求所有的决策单元式整数。

(c) > c.lp <- lp(objective.in = c(1, -3, 4, 1),

+ const.mat = matrix(c(1, -2, 0, 0,

+ 0, 3, 1, 0,

+ 0, 1, 0, 1), nrow = 3, byrow = TRUE),

+ const.rhs = c(9, 9, 10),

+ const.dir = c(">=", ">=", ">="))

```
> c.lp
Error: status 3
```

这表明结果是非约束的。

第七章习题

1.
```
>x<-c(0.45,0.08,-1.08,0.92,1.65,0.53,0.52,-2.15,-2.2,-0.32,-1.87,-0.16,
+ -0.19,-0.98,-0.2,0.67,0.08, 0.38, 0.76, -0.78)
>y<-c(1.26,0.58,-1,1.07,1.28,-0.33,0.68,-2.22,-1.82,-1.17,-1.54,0.35,
+ -0.23,-1.53,0.16,0.91,0.22,0.44,0.98,-0.98)
> library(quadprog)
> X <- cbind( rep(1, 20), x)
> XX <- t(X) %*% X
> Xy <- t(X) %*% y
> A <- matrix( c(1, -1), ncol = 1)
> b <- 0
> ans<- solve.QP( Dmat = XX, dvec = Xy, Amat = A, bvec = b)
> ans$solution
[1] 0.5314036 0.5314036
```

表明截距等于斜率等于 0.5314036。

3.
```
> A <- matrix( c(1,1,-1,0,0,0,0,1,0,0,1,-1,0,0,1,0,0,0,0,1,-1),
+ nrow=3, byrow=TRUE)
> D <- matrix( c( 0.01, 0.002, 0.002,
+ 0.002, 0.01, 0.002,
+ 0.002, 0.002, 0.01), nrow = 3)
> x <- c( 0.002, 0.005, 0.01)
> b <- c( 1, 0, -0.5, 0, -0.5, 0, -0.5)
> ans <- solve.QP( D, x, A, b, meq = 1)
> ans$solution
[1] 0.0625 0.4375 0.5000
```

5.
```
> D <- matrix( c( 0.01, 0,
+ 0, 0.04), nrow = 2, byrow = TRUE)
> A <- matrix( c( 1, 1, 1,
+ 0, 0, 1), nrow = 2, byrow = TRUE)
> x <- c( 0.005, 0.01)
```

```
> b <- c( 1, 0, 0)
```

(a)
```
> ans <- solve.QP( 0.5 * D, x, A, b, meq = 1)
> ans$solution
[1] 1.0 0.5
```

(b)
```
> ans <- solve.QP( D, x, A, b, meq = 1)
> ans$solution
[1] 1.00 0.25
```