

統計データ解析 R 言語の基礎

吉田朋広 (東京大学)
小池祐太 (東京大学)
村田 昇 (早稲田大学・東京大学)

May 17, 2019

東京大学大学院数理科学研究科
統計データ解析教育研究グループ[°]

1 Rの基本的な操作

1.1 はじめに

まず、はじめに R の概要について述べる。

1.1.1 R 言語

R(または R 言語と呼ばれる) は統計計算のための言語と環境の総称であり、オープンソース・フリーソフトウェア (open source, free software) である。利用の規約は GNU General Public License (GPL) に従うので、その内容について詳しく知りたい場合は

<https://www.gnu.org/licenses/gpl-3.0.en.html> (英語)
<https://www.gnu.org/licenses/gpl-3.0.ja.html> (日本語)

を参照して欲しい。

また、多くの人により開発されている多数のパッケージ (package) によって、様々な機能 (パッケージは関数やデータの集合体と考えればよい) を追加することができる。R の本体、およびパッケージは、開発プロジェクトのサイト R Project (The R Project for Statistical Computing)

<http://www.r-project.org/>

のメニューにある CRAN (The Comprehensive R Archive Network) の中にあるミラーサイト (mirror site; 日本国内にもある) からダウンロードすることができる。R の本体は OS (Operating System; Linux, MacOS, Windows) 別に異なる配布物として公開されており、それぞれの OS に適切な方法で簡単にインストールすることができる。また、パッケージは R の中に用意された関数や GUI (Graphical User Interface; 画面上のグラフィックスとマウスなどを用いて直感的な操作を提供するユーザインターフェース) を用いてインストールすることができる。

R Project で公開されている R 本体には Windows や MacOS の場合は専用の GUI が用意されているが、UNIX 系 OS の場合はターミナル (シェル) から起動する必要がある。このため OS によって若干操作性が異なるという問題があるが、UNIX も含め様々な OS において同様に利用することができる RStudio という統合開発環境 (integrated development environment; IDE) が RStudio 社により開発され公開されている。

<https://www.rstudio.com/>

講義では OS による操作の違いができるだけ少なくするために、RStudio を用いて説明を行う。

演習 1.1. R と RStudio を自身の PC にインストールしてみよう。

<http://www.r-project.org/> (The R Project)
<https://www.rstudio.com/> (RStudio, inc)

例えば以下のサイトがインストールの参考になる。

<http://www.okadajp.org/RWiki/?R%20のインストール>
<http://aoki2.si.gunma-u.ac.jp/R/begin.html>

1 R の基本的な操作

1.1.2 起動と終了

RStudio を起動すると、標準では以下のような 4 ペイン(枠; pane)のウィンドウが立ち上がる。左上がエディタ、左下がコンソール、右上が変数や履歴、右下がグラフィックスやヘルプなどを表示するペインとなる。

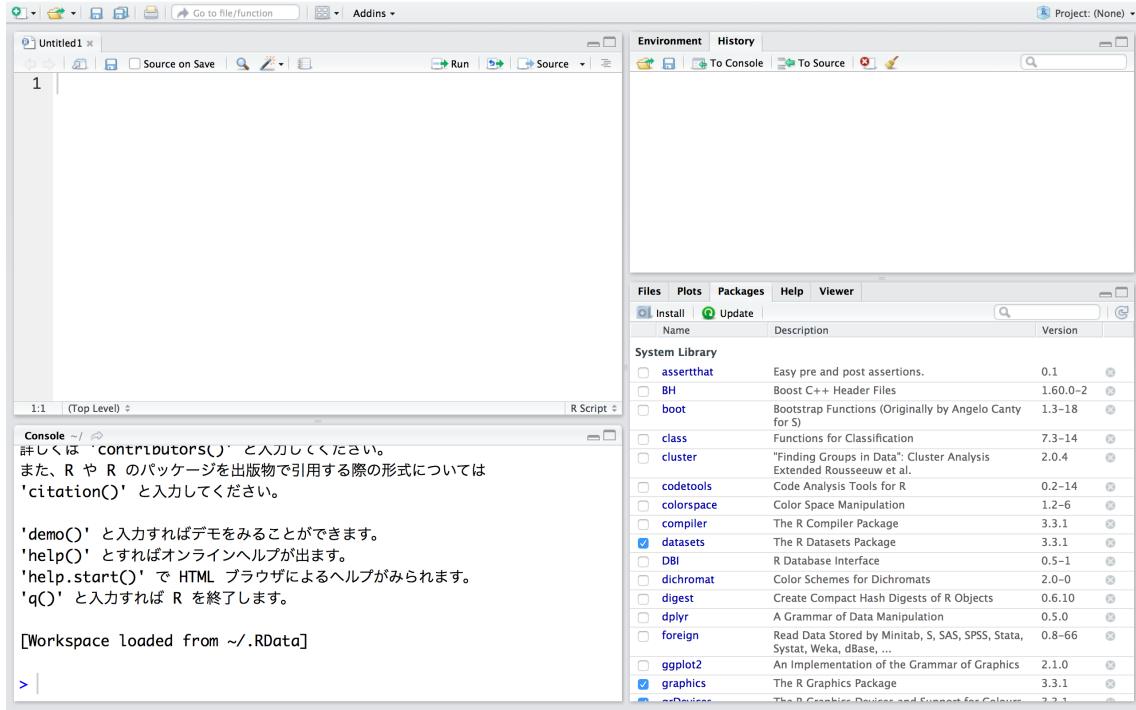


Figure 1.1: RStudio の起動画面。

起動後、コンソールは以下のようなメッセージを表示し、最後に入力を促すプロンプトである ‘>’ 記号を表示して入力待ちの状態となる。

```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R は、自由なソフトウェアであり、「完全に無保証」です。
一定の条件に従えば、自由にこれを再配布することができます。
配布条件の詳細に関しては、「license()」あるいは「licence()」と入力してください。

R は多くの貢献者による共同プロジェクトです。
詳しく述べは「contributors()」と入力してください。
また、R や R のパッケージを出版物で引用する際の形式については
「citation()」と入力してください。

'demo()' と入力すればデモをみることができます。
'help()' とすればオンラインヘルプが出ます。
'help.start()' で HTML ブラウザによるヘルプがみられます。
'q()' と入力すれば R を終了します。

>
```

例えばここで終了を指示する `q()` を入力すれば、R は終了する。

終了時にメッセージが表示される場合があるが、これに対して“y(yes)”を入力すると、それまでに定義された変数や関数およびコマンドの履歴(ヒストリ)が保存され、次回起動時に自動的に読み込まれる。“n(no)”にするとこのセッションで更新した内容は残らない。また、終了することを中止して計算を続ける場合は“c(cancel)”を入力する。

なお、入力文字列において '#' 以降は無視されるので、以降の実行例においては#を用いて必要なコメントを記載していく。

1.1.3 ヘルプ機能

R にはオンラインのヘルプ機能が備えられていて、コンソール(左下のペイン)から関数 `help()` に関数名を、関数 `help.search()` には検索したいキーワードを渡すことによって利用することができる。なお、以下はあくまで一つの出力例であり、環境(R のバージョンやインストールされているパッケージなど)によって出力が異なる場合があることに注意して欲しい。

```
> ### help 関数の使い方
> help(sin) # 三角関数のヘルプを見る
> ?log # help() の代わりに ? を使うことができる
> ### help.search 関数の使い方
> help.search("histogram") # ヒストグラムに関連する関数を探す
> ??random # help.search() の代わりに ?? を使うことができる
```

(basic-help.r)

最初の例は `sin` 関数を調べたもので、右下のペインにヘルプの内容が出る。左上の “Trig” は見出いで、この内容が “trigonometric functions” に関するヘルプであることを表わしている。また中央上の “package:base” は “base” というパッケージ内の関数であることを示している。

二番目の例は “histogram” に関する事項を検索したもので、例えば “graphics::hist” は “graphics” というパッケージ内にある “hist” という関数がヒストグラムの作成に関連することを示している。

なお、上記の例で出てきた “base” や “graphics” は指定しなくても標準で読み込まれるパッケージである。読み込まれているパッケージを確認する方法は次節を参照して欲しい。

GUI を用いる場合は右下のペインの “Help” タブ(tab) を利用する。関数名またはキーワードを入力して必要な情報を検索することができる。

R の本体、あるいはパッケージに関するドキュメント(マニュアル)は開発プロジェクトのサイト CRAN にあるが、使い方を含め有用な情報を解説するサイトとして

```
http://www.okada.jp.org/RWiki/
http://aoki2.si.gunma-u.ac.jp/R/
```

など数多くあるので、これらも合わせて参考して欲しい。

1.1.4 パッケージ管理

CRAN では 2018 年 4 月 3 日現在、12368 を越えるパッケージが公開されている。

右下のペインには “Packages” タブがあり、GUI を用いてパッケージ管理を行うことができる。必要な機能を持つパッケージ名を調べておけば、“Packages” タブの中の “Install” から新規にパッケージをインストールすることができる。また “Update” を選ぶとインストール済のパッケージの更新を行うことができる。なお、標準でいくつかのパッケージは自動的

1 R の基本的な操作

に読み込まれており、既に読み込まれたパッケージは“Packages”タブで確認することができます。

コンソールから直接インストールする場合には、関数 `install.packages()` を用いればよい。以下は一つの出力例であり、環境によっては異なる場合もあることに注意して欲しい。

```
> install.packages("ggplot2",repos='https://cran.ism.ac.jp/')

ダウンロードされたパッケージは、以下にあります
  /var/folders/py/fjm6f39972b0n4__dl9d_8xw0000z8/T//RtmpMsVncS downloaded_packages
>
```

パッケージを取り扱う関数についての更に詳しい情報は `help("install.packages")` や `help("update.packages")`、または `help("INSTALL")` などを利用して調べて欲しい。

1.2 基本的な使い方

1.2.1 式の入力

四則演算や一般的な関数は C 言語などの計算機言語とほぼ同じ名称で使うことができ、直感に沿った文法で計算を実行することができる。

```
> (1 + 3) * (2 + 4) / 6 # 四則演算
[1] 4

> 1.8 + 5 - 0.04 + 8.2 / 3 # 計算順に注意
[1] 9.493333

> pi # π (パイ) は定義されている
[1] 3.141593

> print(pi,digits=22) # 桁数を変更して表示
[1] 3.141592653589793115998

> sqrt(2) # 平方根
[1] 1.414214

> 8^(1/3) # 置乗
[1] 2

> exp(10) # 指数関数
[1] 22026.47

> exp(1) # 自然対数の底
[1] 2.718282

> log(10) # 対数関数 (log, log10, log2)
[1] 2.302585
```

1 R の基本的な操作

```
> sin(pi/2) # 三角関数 (sin, cos, tan)
[1] 1
> sinpi(2/3) # sinpi(x) = sin(pi*x)
[1] 0.8660254
> acos(1/2) # 逆三角関数 (asin, acos, atan)
[1] 1.047198
```

(basic-calc.r)

1.2.2 数の扱い

R では実数および複素数を取り扱うことができ、指数表記にも対応している。また、無限大や不定な数など特殊なものを扱うこともできる。

```
> (1.5+3.5i) * (2-4i) # 複素数の計算 (i の前に数字がある場合のみ虚数とみなす)
[1] 17+1i
> 1i * 1i # 虚数単位は i ではなく 1i
[1] -1+0i
> 1.38e10 * 3.68e-37 / 0.34e-5 # 指数表記の計算
[1] 1.493647e-21
> -log(0) # 無限大 (非常に大きな値)
[1] Inf
> 3 * log(0) # 数として扱える (計算はできる)
[1] -Inf
> sqrt(-1) # Not a Number (非数)
[1] NaN
> sqrt(-1) + 1 # 数として扱えないので計算はできない
[1] NaN
> log(0) / log(0) # これも Not a Number (非数)
[1] NaN
```

(basic-numbers.r)

なお、これらの数値は C 言語にあるような int や double などの数値データの型を気にする必要はない。

1.2.3 変数への代入

文字列を変数名として、数値を保持することができる。また、変数をそのまま計算に用いることもできる。

```
> x <- sin(pi/3) # x に代入
> print(x) # x の値を確認
[1] 0.8660254

> y <- cos(pi/3) # y に代入
> y # print(y) と同じ, y の値を確認
[1] 0.5

> z <- x - y # 計算結果を代入
> (z) # print(z) と同じ, z の値を確認
[1] 0.3660254

> (w <- x * y) # print(w <- x * y) と同じ, 代入結果を表示
[1] 0.4330127

> w # 代入結果を確認 (上と同じ値が表示される)
[1] 0.4330127
```

(basic-variables.r)

変数名は自由に決めて用いることができる(例:x, y, abcなど)。しかし、sin, log, piなどRの仕様として使われているものは、用いることができない訳ではないが混乱を招く元なので使わない方が良い。

なお、Rでは、変数や関数、および関数の実行結果等を総称してオブジェクト(object)と呼ぶ。

演習 1.2. Rを電卓として使ってみよう。

1. 四則演算の計算順を確認する。
2. 複素数の扱い方を確認する。
3. 数学で用いられるどういった関数がRで利用可能か確認する。

1.3 データ構造

Rには、以下のようなデータ構造が用意されている。

- ベクトル (vector)
- 行列 (matrix)
- 配列 (array)
- リスト (list)
- データフレーム (data frame)

1 R の基本的な操作

また、これらのデータは適当な変数を割り当てて保存しておくことができる。

以下ではデータ解析において基本的な役割を担うベクトル、行列、リスト、データフレームについて説明する。

1.3.1 ベクトル

ベクトルはスカラー値の集合(1次元配列)である。

スカラー値として扱われるものには、実数と複素数以外に、文字列、論理値などが含まれる。

```
> ### 実数
> (x <- 4) # 変数 x に実数 4 を代入
[1] 4
> x^10
[1] 1048576
> x^100
[1] 1.606938e+60
> x^1000 # 実数として保持できる最大値を越える
[1] Inf
> ### 複素数
> 1i # "i"の直前に数値を書く
[1] 0+1i
> (1+2i)*(2+1i)
[1] 0+5i
> try(i) # (try を外して確認せよ) iだけでは複素数とみなされずエラーになる
> ### 文字列
> (y <- "foo") # 文字列は ' または " で括る
[1] "foo"
> (z <- "bar") # ("foo" や "bar" は意味のない文字列として良く用いられる)
[1] "bar"
> paste(y,z) # 文字列の足し算、省略時は区切り文字 (separator) は " "(空白)
[1] "foo bar"
> paste(y,z,sep="") # 区切り文字 (separator) を ""(無) に指定
[1] "foobar"
> try(y+z) # (確認せよ) 足し算はできずエラーになる
> ### 論理値
> TRUE # 論理値(真)
[1] TRUE
> T # 論理値(真)の省略形
```

1 R の基本的な操作

```
[1] TRUE
> FALSE # 論理値(偽)
[1] FALSE
> F # 論理値(偽)の省略形
[1] FALSE
> as.numeric(TRUE) # as.numeric は数値に変換する関数
[1] 1
> as.numeric(F)
[1] 0
(basic-scalar.r)
```

一般にベクトルは関数 `c()` を用いて生成することができる。ベクトルの要素を取り出すには `[]` を付けて要素の番号を指定すればよい。

これ以外に規則的な系列を生成するための関数として、等間隔の系列を作るために関数 `seq()`、繰り返しの系列を作るために関数 `rep()` などがある。また、ベクトルの長さを求めるために関数 `length()` が用意されている。

```
> ### 関数 c の使い方
> (x <- c(0,1,2,3,4)) # スカラーを並べてベクトルを作成
[1] 0 1 2 3 4
> (y <- c("foo", "bar")) # 文字列のベクトル
[1] "foo" "bar"
> x[2] # ベクトルの 2 番目の要素
[1] 1
> y[2]
[1] "bar"
> x[c(1,3,5)] # 複数の要素は要素番号のベクトルで指定
[1] 0 2 4
> ### 関数 seq の使い方
> (x <- seq(0,3,by=0.5)) # 0 から 3 まで 0.5 刻みの系列
[1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0
> (y <- seq(0,3,length=5)) # 0 から 3 まで長さ 5 の系列
[1] 0.00 0.75 1.50 2.25 3.00
> (z <- 1:10) # seq(1,10,by=1) と同様
[1] 1 2 3 4 5 6 7 8 9 10
> (z <- 10:1) # 10 から 1 まで 1 刻みの逆順
```

1 R の基本的な操作

```
[1] 10 9 8 7 6 5 4 3 2 1
> z[3:8] # zの3番目から8番目の要素
[1] 8 7 6 5 4 3
> ### 関数 rep の使い方
> (x <- rep(1,7)) # 1を7回繰り返す
[1] 1 1 1 1 1 1 1
> (y <- rep(c(1,2,3),times=3)) # (1,2,3)を3回繰り返す
[1] 1 2 3 1 2 3 1 2 3
> (z <- rep(c(1,2,3),each=3)) # (1,2,3)をそれぞれ3回繰り返す
[1] 1 1 1 2 2 2 3 3 3
> ### その他の操作
> (x <- seq(0,2,by=0.3))
[1] 0.0 0.3 0.6 0.9 1.2 1.5 1.8
> length(x) # ベクトルの長さ
[1] 7
> y <- 2:5
> (z <- c(x,y)) # ベクトルの連結
[1] 0.0 0.3 0.6 0.9 1.2 1.5 1.8 2.0 3.0 4.0 5.0
> rev(z) # rev はベクトルを反転する関数
[1] 5.0 4.0 3.0 2.0 1.8 1.5 1.2 0.9 0.6 0.3 0.0
> LETTERS # アルファベット1文字を要素とするベクトルは定義されている
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"
> letters[1:10] # 小文字の場合
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
(basic-vector.r)
```

1.3.2 行列

一般に行列は関数関数 `matrix()` を用いて生成することができる。行列の (i, j) 成分を取り出すには、`[i, j]` をつければよい。

```
> ### 関数 matrix の使い方
> x <- c(2,3,5,7,11,13) # ベクトルとして定義する
> matrix(x,2,3) # (2,3)行列に変換する
 [,1] [,2] [,3]
[1,]    2     5    11
[2,]    3     7    13
```

1 R の基本的な操作

```
> (X <- matrix(x, ncol=3)) # 列数指定 (行数はそれに合わせて決まる)
 [,1] [,2] [,3]
[1,]    2    5   11
[2,]    3    7   13

> (Y <- matrix(x, ncol=3, byrow=TRUE)) # 横に並べる
 [,1] [,2] [,3]
[1,]    2    3    5
[2,]    7   11   13

> #### その他の操作
> nrow(X) # 行数を取得する
[1] 2

> ncol(X) # 列数を取得する
[1] 3

> X[1,2] # (1,2) 成分を取り出す
[1] 5

> X[2, ] # 2行目を取り出す (列は指定しない)
[1] 3 7 13

> X[,3] # 3列目を取り出す (行は指定しない)
[1] 11 13

> as.vector(X) # ベクトル x に戻る
[1] 2 3 5 7 11 13

> as.vector(Y) # 横に並べた場合はベクトル x に戻らない
[1] 2 7 3 11 5 13

> dim(x) <- c(2,3) # ベクトルに次元属性を与えて行列化する
> x # X と同じ型の行列になる
 [,1] [,2] [,3]
[1,]    2    5   11
[2,]    3    7   13
```

(basic-matrix.r)

1.3.3 リスト

リストは異なる構造のデータをまとめて1つのオブジェクトとして扱えるようにしたものである。リストの各要素は種類がバラバラであってもよい(例えばベクトルと行列が混在していてもよい)。一般にリストは関数 `list()` を用いて作成する。要素を取り出すには `[[[]]]` をつけて要素の番号を指定する。もしくは、各成分に名前をつけることができるので、それを用いて各成分を参照することもできる。

1 R の基本的な操作

```
> ### 関数 list の使い方
> (L1 <- list(c(1,2,5,4), matrix(1:4,2), c("Hello","World")))

[[1]]
[1] 1 2 5 4

[[2]]
[,1] [,2]
[1,]    1    3
[2,]    2    4

[[3]]
[1] "Hello" "World"

> ## 各要素のデータ型はバラバラでよい
> L1[[1]] # リスト L1 の第 1 要素を取り出す

[1] 1 2 5 4

> L1[[2]][2,1] # リストの第 2 要素の (2,1) 成分を取り出す

[1] 2

> L1[[c(3,2)]] # リストの第 3 要素の 2 番目

[1] "World"

> L1[[3]][[2]] # 上と同じ

[1] "World"

> L1[1] # 第 1 要素をリストとして取り出す

[[1]]
[1] 1 2 5 4

> L1[c(1,3)] # リストの複数要素を同時に取り出す

[[1]]
[1] 1 2 5 4

[[2]]
[1] "Hello" "World"

> (L2 <- list(Info="統計データ解析", List=L1)) # 名前付きリストを生成する

$info
[1] "統計データ解析"

$list
$list[[1]]
[1] 1 2 5 4

$list[[2]]
[,1] [,2]
[1,]    1    3
[2,]    2    4

$list[[3]]
[1] "Hello" "World"

> L2[["Info"]] # 要素名で取り出す

[1] "統計データ解析"
```

```
> L2$info # 要素名で取り出す (別記法)
[1] "統計データ解析"
> names(L1) <- c("vector", "matrix", "character") # L1 の要素に名前をつける
> L1 # 変更したリストを表示する
$vector
[1] 1 2 5 4

$matrix
[,1] [,2]
[1,]    1    3
[2,]    2    4

$character
[1] "Hello" "World"
```

(basic-list.r)

1.3.4 データフレーム

データフレームは同じ長さのベクトルを束ねたものであり、解析するデータを纏めた表とを考えることができる。一般にデータフレームは関数 `data.frame()` を用いて作成する。要素を取り出すには `[,]` を付けて要素の行番号・列番号を指定すればよい。また、各行・各列には名前を付けることができるので、それを用いてデータを参照することもできる。

```
> ### 関数 data.frame の使い方
> (x <- data.frame( # 各項目が同じ長さのベクトルを並べる
+   month=c(4,5,6,7),           # 月
+   price=c(900,1000,1200,1100), # 値格
+   deal=c(100,80,50,75)))     # 取引量

  month price deal
1      4    900   100
2      5   1000    80
3      6   1200    50
4      7   1100    75

> x[2,3] # 2行3列を取り出す
[1] 80

> x[3, ] # 3行目を取り出す
  month price deal
3      6   1200    50

> x[ ,2] # 2列目を取り出す
[1] 900 1000 1200 1100

> x$price # 列名で取り出す (上記の別記法)
[1] 900 1000 1200 1100

> x[2]       # 2列目だけからなるデータフレームを取り出す
```

1 R の基本的な操作

```
price
1 900
2 1000
3 1200
4 1100

> x["price"] # 列名で取り出す (上記の別記法)

price
1 900
2 1000
3 1200
4 1100

> x[c("month", "deal")] # 複数列の場合はベクトルで指定する

month deal
1      4   100
2      5    80
3      6    50
4      7    75

> ### 行・列の名前の操作
> rownames(x) # 行の名前を表示する

[1] "1" "2" "3" "4"

> rownames(x) <- c("Apr", "May", "Jun", "Jul") # 行の名前を変更する
> colnames(x) # 列の名前を表示する

[1] "month" "price" "deal"

> colnames(x) <- c("tsuki", "kakaku", "torihiki") # 列の名前を変更する
> x # 変更されたデータフレームを表示する

tsuki kakaku torihiki
Apr      4     900      100
May      5    1000       80
Jun      6    1200       50
Jul      7    1100       75

> x["May", "kakaku"] # 特定の要素を名前で参照する

[1] 1000

(basic-data.frame.r)
```

演習 1.3. 実際のデータに基づいてデータフレームを作成してみよう.

1. 長さの等しいベクトルを作成する.
2. ベクトルを束ねてデータフレームを作成する.
3. データフレームの行・列に適当な名前に変更する.

1.4 補遺

1.4.1 参考文献

確率論、統計学およびRの操作に関する成書は多数あるが、以下を参考として挙げておく。
これ以外にも多数あるので、図書館などで手に取って自分に合ったものを選ばれたい。

1 R の基本的な操作

- [1] 藤澤洋徳. **確率と統計**. 東京: 朝倉書店, 2006.
- [2] 吉田朋広. **数理統計学**. 東京: 朝倉書店, 2006.
- [3] 竹内啓. **数理統計学**. 東京: 東洋経済, 1963.
- [4] 金明哲. **Rによるデータサイエンス(第2版)**. 東京: 森北出版, 2017.
- [5] U. リゲス (石田基広訳). **Rの基礎とプログラミング技法**. 東京: 丸善出版, 2012.
- [6] 奥村晴彦. **Rで楽しむ統計**. 東京: 共立出版, 2016.
- [7] Larry Wasserman. **All of Statistics**. New York: Springer, 2004.
- [8] Gareth James et al. **An Introduction to Statistical Learning with Applications in R**. New York: Springer, 2013.
- [9] 山本義郎, 藤野友和, 久保田貴文. **Rによるデータマイニング入門**. 東京: オーム社, 2015.

1.4.2 亂数の生成

代表的な確率分布に従う乱数の生成を行うことができる。ここでは一様乱数, 正規乱数およびランダムサンプリングを行う関数 `runif()`, `rnorm()`, `sample()` を紹介する。

```
> ### 関数 runif の使い方
> runif(5,min=-1,max=1) # [-1,1] 上の一様乱数を 5 個生成する
[1] -0.9183439 -0.7244030  0.1535345 -0.8100145  0.9434465
> runif(5) # 最大最小について指示がなければ [0,1] 上の一様分布から生成
[1] 0.43015202 0.03824789 0.30870338 0.83430333 0.04203088
> ### 関数 rnorm の使い方
> rnorm(5,mean=3,sd=2) # 平均 3, 標準偏差 2 の正規乱数を 5 個生成する
[1] 5.3393482 2.0804078 3.2584954 0.7024576 1.3432491
> rnorm(5) # 平均と標準偏差について指示がなければ標準正規分布から生成
[1] -0.6398905 -0.1342509 -0.1050045  0.8447779  0.1214377
> ### 関数 sample の使い方
> sample(1:10,size=5) # 1 から 10 の整数からランダムに 5 つ抽出する
[1] 3 5 8 6 1
> sample(1:10,10) # 1 から 10 の整数をランダムに並べ替える。"size=" は省略可
[1] 5 8 9 10 7 1 6 3 4 2
> sample(1:10,10,replace=TRUE) # 1 から 10 の整数から 5 つ復元抽出する
[1] 4 7 1 9 1 8 1 3 10 6
```

(basic-random.r)

これらの関数は数値シミュレーションを行う場合に重要な役割を果たす。

演習 1.4. R を使って乱数を生成してみよう。

1. 計算機で生成される乱数の性質を確認しよう。(ヒント: `help("Random")`)
2. 亂数を複数生成し、そのちらばり具合を確認してみよう。(ヒント: `help("summary")`, `help("hist")`)

2 ベクトルと行列の演算, 関数

データ解析, 多変量解析, パターン認識などで必要となる計算の多くはベクトルと行列を用いた計算である. この章では, これらを R 言語で実現する方法をまとめ.

2.1 ベクトルの計算

まずベクトルのみでのさまざまな計算をまとめ. 以下ではベクトルを太字で, その要素は下付き添字で表現する. 例えば k 次元ベクトルは

$$\mathbf{a} = (a_1, a_2, \dots, a_k) \quad (2.1)$$

のように表す. またベクトル \mathbf{a} の第 i 成分を指す場合には $(\mathbf{a})_i$ のように書くこともある.

2.1.1 和

同じ長さのベクトルの和および差

$$\mathbf{a} \pm \mathbf{b} = (a_1 \pm b_1, a_2 \pm b_2, \dots, a_k \pm b_k) \quad (2.2)$$

は, 数値の和と差のように扱うことができる. 成分による表現では

$$(\mathbf{a} \pm \mathbf{b})_i = a_i \pm b_i$$

と書くことができる.

```
> # ベクトルの加減・スカラー倍
> a <- 1:3 # 長さ 3 のベクトル
> b <- 4:6 # 長さ 3 のベクトル
> a + b # 足し算
[1] 5 7 9
> a - b # もちろん引き算も可
[1] -3 -3 -3
> 2 * a # スカラー倍
[1] 2 4 6
> 2*a - b/3 # 線形結合
[1] 0.6666667 2.3333333 4.0000000
> a + 1:6 # 長さが異なる場合は短い方を繰り返し用いる c(1:3,1:3)+1:6
[1] 2 4 6 5 7 9
> a + 1:5 # 一方の長さがもう一方の長さの整数倍でない場合は警告される
[1] 2 4 6 5 7
```

(vector-sum.r)

2.1.2 積

ベクトルの積は通常内積

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^k a_i b_i \quad (2.3)$$

を指すが、データ解析においては要素毎の積 (Hadamard product, Schur product)

$$\mathbf{a} \circ \mathbf{b} = (a_1 b_1, a_2 b_2, \dots, a_k b_k) \quad (2.4)$$

すなわち

$$(\mathbf{a} \circ \mathbf{b})_i = a_i b_i$$

を計算する場合も多い。2つの意味での積が簡単に計算できるように二項演算子 `%*%` (内積) および `*`(要素毎の積) が定義されている。

```
> a <- 1:3 # 長さ 3 のベクトル
> b <- 4:6
> a %*% b # ベクトルの内積 (計算結果は 1x1 行列)
[1,] [,1]
[1,] 32

> try(a %*% (1:6)) # (確認せよ) 長さが異なるとエラーとなる
> a * b # 要素毎の積 (計算結果はベクトル)

[1] 4 10 18

> a * 1:6 # 長さが異なる場合は足りない方が周期的に拡張される
[1] 1 4 9 4 10 18

> a / b # 除算も成分ごとに計算される
[1] 0.25 0.40 0.50
```

(vector-prod.r)

2.1.3 初等関数の適用

ベクトルに初等関数 (`sin`, `exp`, ...) を適用すると、成分ごとに計算した結果が返される。例えば、ベクトル \mathbf{a} に関数 `sin` を適用した結果は

$$\sin(\mathbf{a}) = (\sin(a_1), \dots, \sin(a_k))$$

となる。

```
> a <- (1:6) * pi/2 # 長さ 6 のベクトル
> sin(a) # 数値誤差のため正確に 0 とならない成分がある
[1] 1.000000e+00 1.224647e-16 -1.000000e+00 -2.449294e-16 1.000000e+00
[6] 3.673940e-16

> exp(a)
```

2 ベクトルと行列の演算, 関数

```
[1]      4.810477   23.140693   111.317778   535.491656   2575.970497
[6] 12391.647808

> log(a)
[1] 0.4515827 1.1447299 1.5501950 1.8378771 2.0610206 2.2433422

(vector-fun.r)
```

演習 2.1. ベクトルの計算をしてみよう.

1. ベクトルの内積から 2 つのベクトルがなす角を求めよ.
2. 2 次元および 3 次元ベクトルの積としては, これら以外に“外積”がある. どのように計算すればよいか調べよ.

2.2 行列とその演算

次に行列のみでのさまざまな計算をまとめる. 以下では行列を大文字で, その要素は下付き添字で表現する. 例えば $m \times n$ 行列は

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \quad (2.5)$$

のように表す. また, 行列 A の (i, j) 成分を指す場合には $(A)_{ij}$ のように書くこともある.

2.2.1 和

同じ大きさの行列の和および差

$$(A \pm B)_{ij} = a_{ij} \pm b_{ij} \quad (2.6)$$

は, ベクトルと同じように記述することができる.

```
> (A <- matrix(1:6,nrow=2,ncol=3))    # 2x3 行列の作成
[,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> (B <- rbind(c(2,3,5),c(7,11,13)))  # 行ベクトル (row) を連結する
[,1] [,2] [,3]
[1,]    2    3    5
[2,]    7   11   13

> (C <- cbind(c(0,0),c(0,1),c(1,0))) # 列ベクトル (column) を連結する
[,1] [,2] [,3]
[1,]    0    0    1
[2,]    0    1    0
```

```
> A + B - C
 [,1] [,2] [,3]
[1,]    3    6    9
[2,]    9   14   19
(matrix-sum.r)
```

2.2.2 積

行列の積

$$(AB)_{ij} = \sum_{k=1}^m a_{ik}b_{kj} \quad (2.7)$$

は、左側の行列の行ベクトルと右側の行列の列ベクトルの内積を各要素とする行列となるので、左側の行列の行数と右側の行列の列数が一致する場合のみ定義される。この積は二項演算子 `%*%` を用いて計算する。なお、行列の転置 (transpose) は関数 `t()` を用いて計算することができ、ある行列とその転置行列の積が簡単に計算できる。これは分散などの計算に活躍する。

一方、ベクトルと同様に同じ大きさの行列の要素毎の積 (Hadamard product, Schur product)

$$(A \circ B)_{ij} = a_{ij}b_{ij} \quad (2.8)$$

も簡単に計算できるようになっており、これは二項演算子 `*` を用いて計算する。

```
> (A <- matrix(1:6,nrow=2,ncol=3)) # 行列の作成 (2x3 行列)
 [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> (B <- rbind(c(2,3,5),c(7,11,13))) # 行ベクトル (row) を連結する (2x3 行列)
 [,1] [,2] [,3]
[1,]    2    3    5
[2,]    7   11   13
> (C <- cbind(c(2,3,5),c(7,11,13))) # 列ベクトル (column) を連結する (3x2 行列)
 [,1] [,2]
[1,]    2    7
[2,]    3   11
[3,]    5   13
> A * B # 要素毎の積
 [,1] [,2] [,3]
[1,]    2    9   25
[2,]   14   44   78
> A / B # 除算も成分ごと
 [,1]      [,2]      [,3]
[1,] 0.5000000 1.0000000 1.0000000
[2,] 0.2857143 0.3636364 0.4615385
> A %*% C # 行列の積 (結果は 2x2 行列)
```

2 ベクトルと行列の演算, 関数

```
[,1] [,2]
[1,] 36 105
[2,] 46 136

> C %*% B # 行列の積 (結果は 3x3 行列)

[,1] [,2] [,3]
[1,] 53 83 101
[2,] 83 130 158
[3,] 101 158 194

> A %*% t(A) # 行列 A とその転置行列の積 (結果は 2x2 行列)

[,1] [,2]
[1,] 35 44
[2,] 44 56
```

(matrix-prod.r)

2.2.3 初等関数の適用

ベクトルの場合と同様に、行列に初等関数 (\sin , \exp , ... など) を適用すると、成分ごとに計算した結果が返される。例えば、行列 A に関数 \sin を適用した結果は

$$\sin(A) = \begin{pmatrix} \sin(a_{11}) & \sin(a_{12}) & \dots & \sin(a_{1n}) \\ \sin(a_{21}) & \sin(a_{22}) & \dots & \sin(a_{2n}) \\ \vdots & & \ddots & \vdots \\ \sin(a_{m1}) & \sin(a_{m2}) & \dots & \sin(a_{mn}) \end{pmatrix}$$

で与えられ、行列 A と同じサイズの行列となる。

```
> A <- matrix((1:6)*pi/2, 2, 3) # 行列の作成 (2x3 行列)
> sin(A)

[,1]          [,2]          [,3]
[1,] 1.000000e+00 -1.000000e+00 1.000000e+00
[2,] 1.224647e-16 -2.449294e-16 3.67394e-16

> exp(A)

[,1]          [,2]          [,3]
[1,] 4.810477 111.3178 2575.97
[2,] 23.140693 535.4917 12391.65

> log(A)

[,1]          [,2]          [,3]
[1,] 0.4515827 1.550195 2.061021
[2,] 1.1447299 1.837877 2.243342
```

(matrix-fun.r)

2.2.4 行列式とトレース

行列に特有な量として行列式とトレース(対角成分の総和)があるが、行列式は関数 `det()` を用いて計算することができる。一方、トレースは専用の関数は用意されていないが、対角

2 ベクトルと行列の演算, 関数

成分を取り出す関数 `diag()` とベクトルの和を計算する関数 `sum()` を用いて簡単に計算できる。

```
> (A <- matrix(1:9,nrow=3,ncol=3)) # 行列の作成 (3x3 行列)
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> det(A) # 行列式 (determinant) の計算
[1] 0

> sum(diag(A)) # トレース (trace) の計算
[1] 15
```

(matrix-det.r)

2.2.5 逆行列

正方行列の逆行列 (inverse matrix) を求めるには関数 `solve()` を用いる。

```
> (A <- matrix(c(2,3,5,7,11,13,17,19,23),nrow=3,ncol=3)) # 正則な正方行列
      [,1] [,2] [,3]
[1,]    2    7   17
[2,]    3   11   19
[3,]    5   13   23

> (B <- solve(A)) # 逆行列の計算
      [,1]          [,2]          [,3]
[1,] -0.07692308 -0.7692308  0.69230769
[2,] -0.33333333  0.5000000 -0.16666667
[3,]  0.20512821 -0.1153846 -0.01282051

> A %*% B # AB = BA = E(単位行列) となることを確認する
      [,1]          [,2]          [,3]
[1,]  1.000000e+00 4.163336e-17 -1.214306e-17
[2,] -5.551115e-17 1.000000e+00 -9.194034e-17
[3,] -2.775558e-16 6.938894e-17  1.000000e+00

> B %*% A
      [,1]          [,2]          [,3]
[1,]  1.000000e+00 8.881784e-16 -8.881784e-16
[2,] -2.220446e-16 1.000000e+00 -2.220446e-16
[3,]  3.642919e-17 1.474515e-16  1.000000e+00
```

(matrix-inv.r)

演習 2.2. 行列の計算をしてみよう。

1. 単位行列と成分が全て 1 の行列を作成せよ。

2. 適当な 2 次正方行列 A に対して, Hamilton-Cayley の定理

$$A^2 - \text{tr}(A)A + \det(A)E_2 = O$$

の成立を確認せよ. ただし E_2 は 2 次単位行列, O は 2 次正方零行列であり, また $\text{tr}(A), \det(A)$ はそれぞれ A のトレース, 行列式を表す.

2.3 ベクトルと行列の計算

2.3.1 行列とベクトルの積

R 言語においては, 列ベクトル・行ベクトルという区別はなく, どちらも同じベクトルとして扱われる. 行列とベクトルの積においては, 行列のどちらからベクトルを掛けるかによって自動的に列ベクトルか行ベクトルが判断されて扱われる. なお, ベクトルも行列の一種であるから, 計算結果は行列として表現されることに注意する.

```
> (A <- matrix(1:16, nrow=4, ncol=4))
      [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16

> (b <- rnorm(4))
[1] 0.00287342 0.47646109 1.79327093 0.32216602

> A %*% b # 列ベクトルとして計算
      [,1]
[1,] 22.71278
[2,] 25.30755
[3,] 27.90232
[4,] 30.49709

> b %*% A # 行ベクトルとして計算
      [,1]      [,2]      [,3]      [,4]
[1,] 7.624272 18.00336 28.38244 38.76153

                                         (linear-calc.r)
```

2.3.2 連立方程式

データ解析の様々な場面で連立一次方程式が現れるが, これは行列とベクトルで表現される. 逆行列の計算に用いた関数 `solve()` の引数として行列とベクトルを与えることによって連立一次方程式を解くことができる.

```
> ### 連立一次方程式
> (A <- matrix(c(2,3,5,7,11,13,17,19,23),3,3)) # 3x3 行列
      [,1] [,2] [,3]
[1,]    2    7   17
```

```
[2,]    3    11   19
[3,]    5    13   23

> (b <- c(2,3,5))
[1] 2 3 5

> (x <- solve(A,b)) # Ax = b を解く
[1] 1.000000e+00 -2.220446e-16 5.124106e-17

> A %*% x # Ax が b と一致するか確認する
[,1]
[1,]    2
[2,]    3
[3,]    5

> #### 行列方程式 (A が同じ複数の連立一次方程式と考えられる)
> (B <- matrix(c(2,1,1,1),2,2)) # 2x2 行列
[,1] [,2]
[1,]    2    1
[2,]    1    1

> (C <- matrix(c(4,3,10,7),2,2)) # 2x2 行列
[,1] [,2]
[1,]    4   10
[2,]    3    7

> (X <- solve(B,C)) # BX = C を解く
[,1] [,2]
[1,]    1    3
[2,]    2    4

> B%*%X # BX が C と一致するか確認する
[,1] [,2]
[1,]    4   10
[2,]    3    7

(linear-eq.r)
```

演習 2.3. 行列とベクトルの計算をしてみよう.

1. 適当な 2 次元のベクトルを 120 度回転しなさい.
2. $n \times n$ 行列 A と n 次のベクトル b を作成し, $A \%* \%b + b \%* \%A$ を計算せよ (エラーになる). 何故, そうなるか理由を考えなさい.
3. 連立方程式の問題を作成し, それを解きなさい.

2.4 関数

2.4.1 制御文

一般に最適化や数値計算などを行うためには, 条件分岐や繰り返しを行うための仕組みが必要となる. 多くの計算機言語では `if`(条件分岐), `for`・`while`(繰り返し)を用いた構文が用

意されているが、これを制御文と言う。R言語においてもこれらの構文は用意されており、制御文を使うことによって次節で述べるような複雑な計算を行う関数を定義することができる。

```
> ### 条件分岐 (if)
> x <- 5
> if(x > 0) { # 正か否か判定
+   ## 条件が真の場合に実行するブロック
+   print("positive")
+ } else {
+   ## 条件が偽の場合に実行するブロック
+   print("negative")
+ }

[1] "positive"

> ## else 以下はなくとも動く
> if(x > 0) {
+   print("positive")
+ }

[1] "positive"

> ## 評価が簡単な場合の条件分岐 (ifelse)
> ifelse(x < 0, "true", "not true")

[1] "not true"

> ### 繰り返し (for)
> y <- 0
> for(i in 1:10) { # 1-10 の合計を計算
+   y <- y + i
+ }
> print(y)

[1] 55

> ### 繰り返し (while)
> z <- 1
> n <- 0
> while(z < 100) { # 100未満の間は2倍し続ける
+   z <- 2 * z
+   n <- n + 1
+ }
> print(z) # 100を超えた際のzの値

[1] 128

> print(n) # 条件を満たすまでの回数

[1] 7
```

(fun-control.r)

2.4.2 関数の定義

一般に関数とは入力を規則に従って変換し出力する仕組みを指す。Rでは、入力を引数(argument), 出力を返値(value)と呼び、関数 `function()` を用いて自由に関数を定義することができます。

```

> ### 階乗を計算する関数
> fact <- function(n){ # 素直に計算
+   ifelse(n>0,prod(1:n),1)
+ }
> fact2 <- function(n){ # 再帰的に定義
+   if(n>0) {
+     return(n*fact2(n-1)) # 自分を呼び出す
+   } else {
+     return(1) # fact2(0) = 0! = 1
+   }
+ }
> fact(10) # 同じ結果になるか確認する
[1] 3628800
> fact2(10)
[1] 3628800

```

(fun-define.r)

同じ機能を持つ関数でも定義の仕方はいろいろ工夫できる。

演習 2.4. 以下の機能を持つ関数を作つてみよう。

1. 2次方程式の3つの係数を入力すると解を出力する関数を作成しなさい。
2. 第1項, 第2項, および項数を入力すると Fibonacci 数列を指定された項数まで出力する関数を作成しなさい。なお, Fibonacci 数列とは下記の漸化式を満たす数列である。

$$a_n = a_{n-1} + a_{n-2}, n = 3, 4, \dots$$

3. 自身で仕様を決めて, それを満たす関数を作成しなさい。

2.5 補遺

2.5.1 参考文献

この章の内容に関連する参考書として以下を挙げておく。

- [1] 藤澤洋徳. **確率と統計**. 東京: 朝倉書店, 2006.
- [2] 吉田朋広. **数理統計学**. 東京: 朝倉書店, 2006.
- [3] 竹内啓. **数理統計学**. 東京: 東洋経済, 1963.
- [4] 金明哲. **Rによるデータサイエンス(第2版)**. 東京: 森北出版, 2017.
- [5] U. リゲス (石田基広訳). **Rの基礎とプログラミング技法**. 東京: 丸善出版, 2012.
- [6] 奥村晴彦. **Rで楽しむ統計**. 東京: 共立出版, 2016.
- [7] Larry Wasserman. **All of Statistics**. New York: Springer, 2004.
- [8] Gareth James et al. **An Introduction to Statistical Learning with Applications in R**. New York: Springer, 2013.
- [9] 山本義郎, 藤野友和, 久保田貴文. **Rによるデータマイニング入門**. 東京: オーム社, 2015.

2.5.2 ベクトルのノルム

通常の l^2 ノルム

$$\|\mathbf{a}\|_2 = \sqrt{\sum_{i=1}^k |a_i|^2} \quad (2.9)$$

は内積を用いれば計算できる。一般の l^p ノルム

$$\|\mathbf{a}\|_p = \left(\sum_{i=1}^k |a_i|^p \right)^{1/p} \quad (2.10)$$

や l^∞ ノルム

$$\|\mathbf{a}\|_\infty = \max_{1 \leq i \leq k} |a_i| \quad (2.11)$$

は、要素の和を計算する関数 `sum()`、および最大値を取り出す関数 `max()` を利用して計算することができる。

```
> (a <- rnorm(6)) # 標準正規乱数でベクトルを作成
[1] -0.1183221  1.2478725  0.3633840  1.1183345 -1.2284541 -0.6494950
> sqrt(a %*% a) # l2 ノルム (計算結果は行列型で表示される)
[1,] [,1]
[1,] 2.210169
> as.vector(sqrt(a %*% a)) # l2 ノルム (行列をベクトル型に変換して表示)
[1] 2.210169
> sum(abs(a)) # l1 ノルム
[1] 4.725862
> sum(abs(a)^2)^^(1/2) # lp ノルム (p=2)
[1] 2.210169
> sum(abs(a)^3)^^(1/3) # lp ノルム (p=3)
[1] 1.767239
> max(abs(a)) # 最大ノルム
[1] 1.247872
(vector-norm.r)
```

2.5.3 行列のノルム

行列のノルムにはいくつか定義があるが、関数 `norm()` によって作用素ノルム ($p = 1$ および $p = \infty$),

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^m |a_{ij}| \quad (2.12)$$

$$\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^n |a_{ij}| \quad (2.13)$$

Frobenius ノルム,

$$\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2} \quad (2.14)$$

最大ノルム,

$$\|A\|_{\max} = \max\{|a_{ij}|\} \quad (2.15)$$

およびスペクトルノルム

$$\|A\|_2 = \sigma_{\max}(A), \quad (A \text{ の最大特異値}) \quad (2.16)$$

を計算することができる。

```
> (A <- matrix(c(1,0,0,0,2,0,1,-1,-3,0,0,0),nrow=3,ncol=4)) # 3x4 行列
      [,1] [,2] [,3] [,4]
[1,]     1     0     1     0
[2,]     0     2    -1     0
[3,]     0     0    -3     0

> norm(A,type="1") # 作用素ノルム (p=1; one norm)
[1] 5

> norm(A,type="I") # 作用素ノルム (p=infinity; infinity norm)
[1] 3

> norm(A,type="F") # Frobenius ノルム (Frobenius norm)
[1] 4

> norm(A,type="M") # 最大ノルム (max norm)
[1] 3

> norm(A,type="2") # スペクトルノルム (spectral norm; 2-norm)
[1] 3.408689
```

(matrix-norm.r)

2.5.4 一般化逆行列

データ解析においては、正方でない行列の一般化逆行列 (擬似逆行列; pseudo-inverse matrix) がしばしば必要となる。一般化逆行列 A^\dagger とは

$$AA^\dagger A = A \quad (2.17)$$

2 ベクトルと行列の演算, 関数

が成り立つ行列のことである。いくつかの定義がある。一般化逆行列の中で良く用いられるのは Moore-Penrose の一般化逆行列 (Moore-Penrose pseudoinverse) と呼ばれるもので、これを A^+ と書くことにすると

$$AA^+A = A \quad (2.18)$$

$$A^+AA^+ = A^+ \quad (2.19)$$

$$(AA^+)^* = AA^+ \quad (* \text{は随伴行列の意}) \quad (2.20)$$

$$(A^+A)^* = A^+A \quad (2.21)$$

が成立する行列である。ただし、随伴行列とは、転置かつ複素共役をとった行列のことである。これはパッケージ MASS の中の関数 `ginv()` を用いて求めることができる。

```
> ### 一般化逆行列
> require(MASS) # MASS パッケージを読み込む
> (C <- matrix(rnorm(6), nrow=2, ncol=3)) # ランダムに 2x3 行列を作成
      [,1]      [,2]      [,3]
[1,] -0.6592804 -0.2907673  0.3300182
[2,] -1.4226641  0.4391958 -1.7654877

> (D <- ginv(C)) # 一般化逆行列の計算
      [,1]      [,2]
[1,] -0.9679487 -0.2254243
[2,] -0.5004990  0.1036975
[3,]  0.6554840 -0.3589679

> C %*% D %*% C # CC^+ + C = C であることを確認
      [,1]      [,2]      [,3]
[1,] -0.6592804 -0.2907673  0.3300182
[2,] -1.4226641  0.4391958 -1.7654877

(matrix-ginv.r)
```

2.5.5 固有値と固有ベクトル

一般に n 次正方形行列 A に対して、複素数 λ と零ベクトルでない n 次元ベクトル x が

$$Ax = \lambda x$$

を満たすとき、 λ を A の固有値、 x を λ に対する固有ベクトルと呼ぶ。固有値および固有ベクトルはデータ解析にしばしば用いられ、R では関数 `eigen()` で求めることができる。

```
> (A <- matrix(c(1,-1,-1,1), 2, 2)) # 2x2 行列
      [,1] [,2]
[1,]    1   -1
[2,]   -1    1

> r <- eigen(A) # 結果は固有値と固有ベクトルからなるリスト
> r$values # 固有値

[1] 2 0
```

```
> r$vectors # 固有ベクトルからなる行列
      [,1]      [,2]
[1,] -0.7071068 -0.7071068
[2,]  0.7071068 -0.7071068

> ## r$vectors[,i] が r$values[i] に対する固有ベクトルに対応
> t(r$vectors) %*% A %*% r$vectors # 対角化

      [,1]      [,2]
[1,]    2 -4.474229e-17
[2,]    0  0.000000e+00

(matrix-eigen.r)
```

2.5.6 特異値分解

大規模データに対するデータ解析では、正方行列でない任意の行列に対する分解が必要となることがある。一般に実 $n \times p$ 行列 A に対して、 $q = \min\{n, p\}$ とすると、実 $n \times q$ 行列 U 、実 $p \times q$ 行列 V 、非負 q 次対角行列 D が存在して、 $U^T U = V^T V = E_q$ を満たし、かつ

$$A = UDV^T \quad (2.22)$$

と書けることが知られている。ただし T は転置の意で、 E_q は q 次単位行列を表す。この分解を A の特異値分解と呼び、 D の対角成分を A の特異値と呼ぶ。また、行列 A の特異値は A から一意的に定まることが知られている。特異値分解は関数 `svd()` で実行できる。

```
> (A <- matrix(1:6, nrow=2)) # 2x3 行列
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> s <- svd(A) # 結果は特異値と行列 U, V からなるリスト
> s$d # 特異値
[1] 9.5255181 0.5143006

> s$u # 行列 U
      [,1]      [,2]
[1,] -0.6196295 -0.7848945
[2,] -0.7848945  0.6196295

> s$v # 行列 V
      [,1]      [,2]
[1,] -0.2298477  0.8834610
[2,] -0.5247448  0.2407825
[3,] -0.8196419 -0.4018960

> s$u %*% diag(s$d) %*% t(s$v) # 行列 A の再現
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

(matrix-svd.r)

演習 2.5. 以下の計算をしてみよう.

1. 関数 `ginv()` で計算される行列が Moore-Penrose の一般化逆行列になっていることを確かめよ.
2. 行列 A の Frobenius ノルムを, $A * A$ および $A \% * \%t(A)$ を用いて計算しなさい. (ヒント: $\|A\|_F^2 = \text{tr } AA^T$)
3. A が非負定値 n 次対称行列, すなわち固有値がすべて非負の n 次対称行列のとき, A の対角化を考えることで $A = B^2$ を満たす非負定値 n 次対称行列 B が求められる. この行列 B を計算するプログラムを作成せよ.

3 データの加工・整理と入出力

収集されたデータを整理して実際の解析を行うためには、特定の条件に当て嵌まる行や列の選択、複数の量から計算した統計量によって新たな行を作成、データをグループ化して集計など、様々な操作が要請される。以下では、基本的なデータの型とその操作について解説する。

3.1 データの形式

まず、Rで用いられる基本的なデータ形式についてもう一度まとめておく。

3.1.1 値の型

Rでは数値、文字列、真偽値を扱うことができる。どの型であるかを確認する場合には関数 `mode()` や関数 `typeof()`(数値の型などについて少し詳しい)を用いればよい。

```
> ### 値の型
> (a <- 2.718)      # 数値 (実数)
[1] 2.718
> mode(a)           # 型を確認
[1] "numeric"
> typeof(a)          # 型を確認 (内部での扱いを表すので詳しい分類が示される)
[1] "double"
> (b <- 3.5 + 5.8i) # 数値 (複素数)
[1] 3.5+5.8i
> mode(b)           # 型を確認
[1] "complex"
> (c <- "alphabet") # 文字列
[1] "alphabet"
> mode(c)           # 型を確認
[1] "character"
> (d <- FALSE)       # 真偽値 (TRUE/FALSE)
[1] FALSE
> mode(d)           # 型を確認
[1] "logical"
```

(`data-type.r`)

3.1.2 ベクトル

1つ、または複数の値を並べたものがベクトルである。規則的な系列を生成するためには関数 `seq()` や関数 `rep()` など、いろいろな方法が用意されている。ベクトルの長さを知るためには関数 `length()` を用いる。

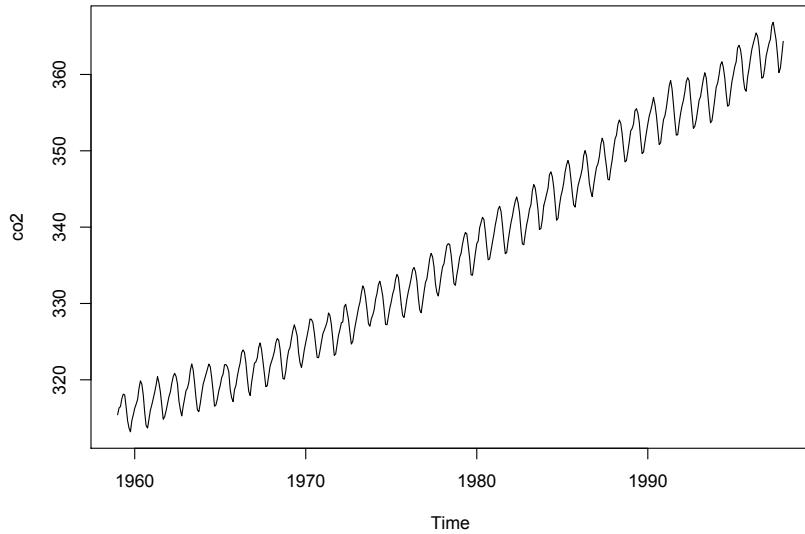


Figure 3.1: ベクトルデータの表示の例

[Figure 3.1 を参照]

```
> ### ベクトルの例
> (a <- c(2,3,5,7,11)) # 要素を指定
[1] 2 3 5 7 11
> (b <- 7.5:1) # 差 1 の等差数列 (seq(7.5, 1, by=-1) と同じ)
[1] 7.5 6.5 5.5 4.5 3.5 2.5 1.5
> (c <- rep(c(2,3),length.out=7)) # 長さ 7 の 2,3 の繰り返し
[1] 2 3 2 3 2 3 2
> ### データ例 datasets::co2 (詳細は help(co2) を参照)
> length(co2) # 長さを確認
[1] 468
> head(co2,n=8) # 最初の 8 個を表示
[1] 315.42 316.31 316.50 317.56 318.13 318.00 316.39 314.65
> plot(co2) # グラフ表示 (ベクトルを拡張して時系列の情報が入っている)
```

(data-vector.r)

3.1.3 行列

行列は2次元状に値を並べたものであり、次節で説明する配列の2次元版と考えることができます。行列は、関数`matrix()`によって1つのベクトルを並べ替える、あるいは関数`cbind()`や関数`rbind()`によって複数のベクトルを連結して作成することができます。行列の次元(サイズ)を知るために関数`dim()`を用い、行数および列数を求めるにはそれぞれ関数`nrow()`、関数`ncol()`を用いる。また、関数`rownames()`および関数`colnames()`を用いて行と列に名前を付けることができる。

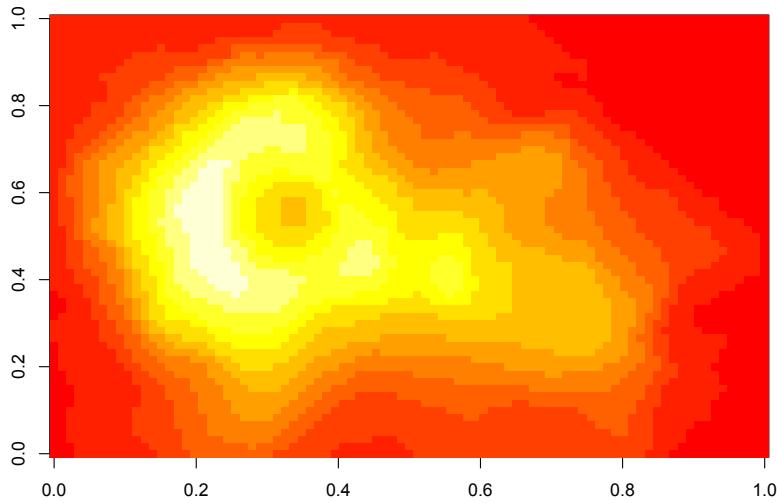


Figure 3.2: 行列データの表示の例

[Figure 3.2 を参照]

```
> ### 行列の例 (ベクトルから行列を作成)
> (A <- matrix(1:6, nrow=2, ncol=3)) # 列ごとに並べる
 [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> (B <- cbind(c(1,2),c(3,4),c(5,6))) # 列で結合する
 [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> (C <- matrix(1:6, nrow=2, ncol=3, byrow=TRUE)) # 行ごとに並べる
 [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6

> (D <- rbind(c(1,2,3),c(4,5,6))) # 行で結合する
 [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

```

> dim(D) # 大きさを確認
[1] 2 3
> (rownames(D) <- LETTERS[1:nrow(D)]) # 行に名前を付ける
[1] "A" "B"
> (colnames(D) <- letters[1:ncol(D)]) # 列に名前を付ける
[1] "a" "b" "c"
> D # Dの内容を確認する
      a b c
A 1 2 3
B 4 5 6
> ### データ例 datasets::volcano (詳細は help(volcano))
> dim(volcano)      # 大きさを確認
[1] 87 61
> volcano[1:3,1:5] # 左上の3行5列を表示
     [,1] [,2] [,3] [,4] [,5]
[1,] 100 100 101 101 101
[2,] 101 101 102 102 102
[3,] 102 102 103 103 103
> image(volcano)    # 濃淡図として表示

```

(data-matrix.r)

3.1.4 配列

行列を一般化したものとして配列が用意されており、関数 `array()` を用いてベクトルを並べ替え作成することができる。次元を知るためには行列と同様に関数 `dim()` を用いる。

[Figure 3.3 を参照]

```

> ### 配列の例
> (A <- array(1:13, dim=c(3,4,2))) # 3x4x2次の配列
, , 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    3    6    9
[2,]    1    4    7   10
[3,]    2    5    8   11
> ### データ例 datasets::Titanic (詳細は help(Titanic))
> dim(Titanic) # 大きさを確認

```

3 データの加工・整理と入出力

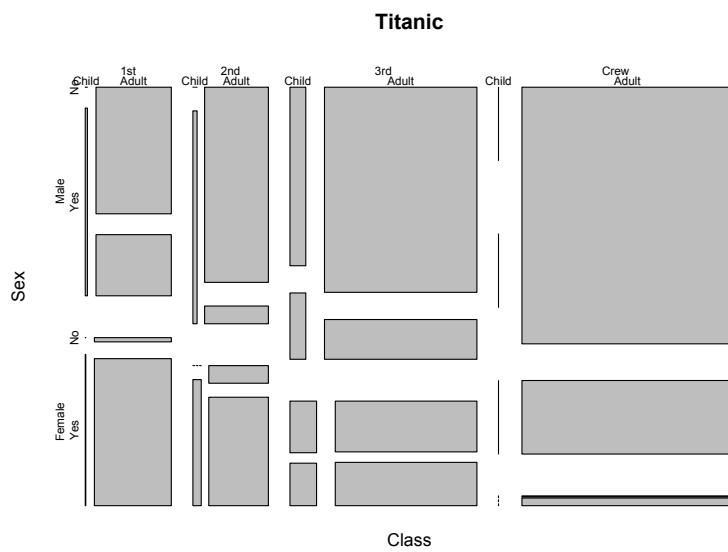


Figure 3.3: 配列データの表示の例

```
[1] 4 2 2 2
> Titanic      # データを表示
, , Age = Child, Survived = No

      Sex
Class  Male Female
1st     0     0
2nd     0     0
3rd    35    17
Crew    0     0

, , Age = Adult, Survived = No

      Sex
Class  Male Female
1st   118     4
2nd   154    13
3rd   387    89
Crew   670     3

, , Age = Child, Survived = Yes

      Sex
Class  Male Female
1st     5     1
2nd    11    13
3rd    13    14
Crew    0     0

, , Age = Adult, Survived = Yes

      Sex
Class  Male Female
```

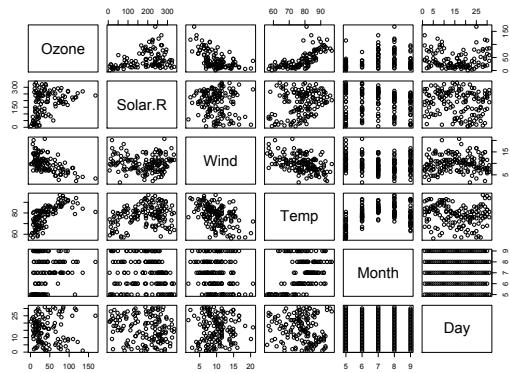
```
1st      57     140
2nd      14      80
3rd      75      76
Crew    192      20
```

> `plot(Titanic) # タイル図として表示`

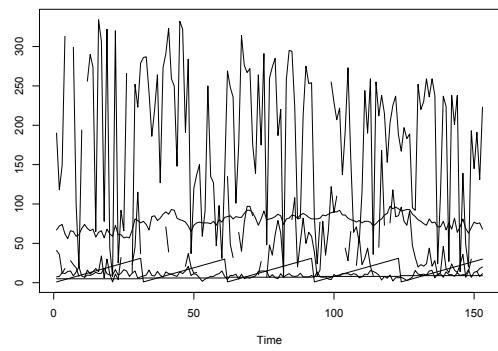
(data-array.r)

3.1.5 データフレーム

データフレームは表を取り扱うためのデータ形式で、行列と同様な2次元の配列と考えることができる。さまざまな形式のデータを変換してデータフレームを作成するには、関数 `data.frame()` によってベクトルを連結して作成する、あるいは行列を変換して作成するなど、いくつかの方法が用意されている。また、データフレームの中から必要な部分集合を取り出すために、関数 `subset()` が用意されている。行列と同様に行名、列名をつけることができるが、行名には関数 `rownames()` を、列名には関数 `names()` を用いる。



(a) 散布図



(b) 時系列のグラフ

Figure 3.4: データフレームの表示の例

```
> ### データフレームの例 (ベクトルから行列を作成)
> (a <- data.frame(height=c(172,158,160),weight=c(60,53,51)))

height weight
1     172     60
2     158     53
3     160     51

> (b <- matrix(1:8,nrow=4,ncol=2))

 [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8

> (c <- data.frame(b)) # 行列から作ることもできる
```

3 データの加工・整理と入出力

```
X1 X2
1 1 5
2 2 6
3 3 7
4 4 8

> (rownames(c) <- letters[1:nrow(c)]) # 行名を付ける
[1] "a" "b" "c" "d"

> (names(c) <- c("Left", "Right")) # 列名を付ける
[1] "Left" "Right"

> c # 内容を確認する

  Left Right
a     1      5
b     2      6
c     3      7
d     4      8

> ### データ例 datasets::airquality (詳細は help(airquality))
> dim(airquality)      # 大きさを確認
[1] 153   6

> names(airquality)    # 列の名前を表示
[1] "Ozone"    "Solar.R" "Wind"     "Temp"     "Month"    "Day"

> head(airquality, n=5) # 最初の 5 つのデータを表示

  Ozone Solar.R Wind Temp Month Day
1     41       190 7.4   67     5   1
2     36       118 8.0   72     5   2
3     12       149 12.6  74     5   3
4     18       313 11.5  62     5   4
5     NA        NA 14.3  56     5   5

> plot(airquality)      # 散布図を表示
> ts.plot(airquality)   # 時系列として表示
> subset(airquality, Ozone>100) # 条件を満たす部分集合を抽出

  Ozone Solar.R Wind Temp Month Day
30     115       223 5.7   79     5  30
62     135       269 4.1   84     7   1
86     108       223 8.0   85     7  25
99     122       255 4.0   89     8   7
101    110       207 8.0   90     8   9
117    168       238 3.4   81     8  25
121    118       225 2.3   94     8  29

> subset(airquality, Ozone>100, select=Wind:Day)

  Wind Temp Month Day
30     5.7   79     5  30
62     4.1   84     7   1
86     8.0   85     7  25
99     4.0   89     8   7
101    8.0   90     8   9
117    3.4   81     8  25
121    2.3   94     8  29
```

(data-data.frame.r)

実際のデータ解析においては、より複雑な操作を行う必要もあり、そうした操作に対応するためのパッケージも多数ある。中でも最近よく使われるパッケージ `dplyr` については補遺にて詳しく解説する。

演習 3.1. いろいろな形式のデータを作成してみよう。

1. 関数 `c()`, `seq()`, `rep()`, `matrix()`, `array()`, `data.frame()` などの使い方を、関数 `help()` を用いて調べなさい。
2. データの形式を調べる関数 `is.XXX` や、データの形式を変換する関数 `as.XXX` について調べなさい。
3. 関数 `data()` を用いて、R にどのようなデータ集合が用意されているか調べなさい。

3.2 データの抽出

データから必要な部分集合を取り出すためには、添え字を指定するのが最も基本的な方法である。添え字の指定の仕方には、番号を指定する以外に、論理値で指定する方法がある。この場合、`TRUE` は要素の「選択」を、`FALSE` は要素の「除外」を意味する。また、前にも述べたように、要素に名前が付けられている場合は、その名前によってアクセス可能である。また、マイナス記号をつけて添え字番号を指定すると、その添え字番号の要素を除外する。

```
> ### ベクトルの要素の指定
> x <- c(4, 1, 2, 9, 8, 3, 6)
> x[c(5,2)]      # 5番目と2番目の要素をこの順で抽出
[1] 8 1
> x[-c(2,3,7)]  # 2,3,7番目以外の要素を表示
[1] 4 9 8 3
> (idx <- x>3)  # 3より大きい要素は TRUE, 3以下の要素は FALSE
[1] TRUE FALSE FALSE TRUE TRUE FALSE TRUE
> which(idx)     # TRUEに対応する要素の番号を表示
[1] 1 4 5 7
> x[idx]         # 3より大きい要素 (TRUE) をすべて表示
[1] 4 9 8 6
> x[x>3]         # 上と同じ
[1] 4 9 8 6
> x[which(idx)] # 上と同じ
[1] 4 9 8 6
> x[-c(2,3,7)]  # 2,3,7番目以外の要素を表示
[1] 4 9 8 3
> x[c(2,5)] <- c(0,1) # 2番目と5番目の要素を文字 0 と 1 に置換
> x
```

3 データの加工・整理と入出力

```
[1] 4 0 2 9 1 3 6

> (names(x) <- letters[1:length(x)]) # x の要素にアルファベットを順に名前をつける
[1] "a" "b" "c" "d" "e" "f" "g"

> x[c("b", "e")] # 2番目と5番目の要素
b e
0 1

> ### データフレームの要素の指定
> ### データ例 datasets::airquality (詳細は help(airquality) 参照)
> ### データに NA(欠損) があるので注意
> dim(airquality) # 大きさを確認

[1] 153   6

> names(airquality) # 列名を表示
[1] "Ozone"    "Solar.R"  "Wind"      "Temp"      "Month"     "Day"

> head(airquality) # 最初の6行を表示
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5    1
2    36     118  8.0   72     5    2
3    12     149 12.6   74     5    3
4    18     313 11.5   62     5    4
5    NA      NA 14.3   56     5    5
6    28     NA 14.9   66     5    6

> str(airquality) # オブジェクトの構造を表示
'data.frame': 153 obs. of 6 variables:
 $ Ozone : int 41 36 12 18 NA 28 23 19 8 NA ...
 $ Solar.R: int 190 118 149 313 NA NA 299 99 19 194 ...
 $ Wind : num 7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 $ Temp : int 67 72 74 62 56 66 65 59 61 69 ...
 $ Month : int 5 5 5 5 5 5 5 5 5 5 ...
 $ Day   : int 1 2 3 4 5 6 7 8 9 10 ...

> airquality$Ozone > 100 # Ozone が 100 を超える行を抽出 (NA は NA となる)
[1] FALSE FALSE FALSE FALSE NA FALSE FALSE FALSE FALSE NA FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE NA FALSE FALSE FALSE
[25] NA NA NA FALSE FALSE TRUE FALSE NA NA NA NA NA
[37] NA FALSE NA FALSE FALSE NA NA FALSE NA NA FALSE FALSE
[49] FALSE FALSE FALSE NA NA NA NA NA NA NA NA NA
[61] NA TRUE FALSE FALSE NA FALSE FALSE FALSE FALSE FALSE FALSE NA
[73] FALSE FALSE NA FALSE FALSE FALSE FALSE FALSE FALSE FALSE NA NA
[85] FALSE TRUE FALSE FALSE
[97] FALSE FALSE TRUE FALSE TRUE NA NA FALSE FALSE FALSE NA FALSE
[109] FALSE FALSE FALSE FALSE FALSE FALSE NA FALSE TRUE FALSE NA FALSE
[121] TRUE FALSE FALSE
[133] FALSE FALSE
[145] FALSE FALSE FALSE FALSE FALSE NA FALSE FALSE FALSE

> which(airquality$Ozone > 100) # Ozone が 100 を超える行の番号を抽出
[1] 30 62 86 99 101 117 121

> which(airquality$Ozone > 100 & airquality$Wind <= 5) # 複数の条件の AND
```

3 データの加工・整理と入出力

```
[1] 62 99 117 121
> which(with(airquality, Ozone>100 & Wind<=5))      # 上と同じ
[1] 62 99 117 121
> which(with(airquality, Ozone>100 | Wind<=5))      # 複数の条件の OR
[1] 30 53 54 62 66 86 98 99 101 117 121 126 127
> airquality[which(airquality$Ozone>100), ] # 条件を満たす行を抽出
   Ozone Solar.R Wind Temp Month Day
30     115     223  5.7    79      5   30
62     135     269  4.1    84      7    1
86     108     223  8.0    85      7   25
99     122     255  4.0    89      8    7
101    110     207  8.0    90      8    9
117    168     238  3.4    81      8   25
121    118     225  2.3    94      8   29
> ## airquality[airquality$Ozone>100, ] # 判定結果に NA が含まれるため抽出できない
> airquality[which(airquality$Ozone>100), c("Month", "Day")] # 特定の列のみ表示
   Month Day
30      5  30
62      7  1
86      7  25
99      8  7
101     8  9
117     8  25
121     8  29
```

(data-select.r)

データフレームから必要な部分集合を取り出す際に複雑な条件を指定する場合、添え字を指定するのではコードが読みにくくなってしまう。そのような場合にも対応できるように関数 `subset()` が用意されている。関数 `subset()` の基本的な書式は

```
subset(x, subset, select, drop=FALSE)
```

である。`x` にデータフレームを指定し、`subset` に抽出したい行に関する条件を、`select` に抽出したい列に関する条件をそれぞれ指定し、`drop` は結果が1行または1列のデータフレームになる場合にベクトルとして返すか否かを指定するオプションである。

```
> #### 関数 subset の使い方
> #### subset(dataframe, subset=条件, select=列)
> ## Ozone が 100 を超える行を抽出
> subset(airquality, subset = Ozone>100)

   Ozone Solar.R Wind Temp Month Day
30     115     223  5.7    79      5   30
62     135     269  4.1    84      7    1
86     108     223  8.0    85      7   25
99     122     255  4.0    89      8    7
101    110     207  8.0    90      8    9
117    168     238  3.4    81      8   25
121    118     225  2.3    94      8   29
```

3 データの加工・整理と入出力

```
> ## Ozone が 100 を超える行で列名が Wind と Day のデータ
> subset(airquality, Ozone>100, select=c(Wind,Day)) #

  Wind Day
30   5.7 30
62   4.1  1
86   8.0 25
99   4.0  7
101  8.0  9
117  3.4 25
121  2.3 29

> ## Ozone が 100 を超える行で列名が Wind から Day までのデータ
> subset(airquality, Ozone>100, select=Wind:Day) #

  Wind Temp Month Day
30   5.7    79      5 30
62   4.1    84      7  1
86   8.0    85      7 25
99   4.0    89      8  7
101  8.0    90      8  9
117  3.4    81      8 25
121  2.3    94      8 29

> ## Ozone に欠測 (NA) がなく、かつ Day が 1 か 2 のデータ (AND)
> subset(airquality, !is.na(Ozone) & Day %in% c(1, 2))

  Ozone Solar.R Wind Temp Month Day
1     41       190  7.4   67      5  1
2     36       118  8.0   72      5  2
62    135       269  4.1   84      7  1
63    49        248  9.2   85      7  2
93    39        83   6.9   81      8  1
94     9        24  13.8   81      8  2
124   96       167  6.9   91      9  1
125   78       197  5.1   92      9  2

> ## Ozone が 100 以上か、または Wind が 5 以下のデータ (OR)
> subset(airquality, Ozone>100 | Wind <= 5)

  Ozone Solar.R Wind Temp Month Day
30    115      223  5.7   79      5 30
53     NA       59   1.7   76      6 22
54     NA       91   4.6   76      6 23
62    135      269  4.1   84      7  1
66     64      175  4.6   83      7  5
86    108      223  8.0   85      7 25
98     66       NA  4.6   87      8  6
99    122      255  4.0   89      8  7
101   110      207  8.0   90      8  9
117   168      238  3.4   81      8 25
121   118      225  2.3   94      8 29
126   73       183  2.8   93      9  3
127   91       189  4.6   93      9  4

> ## Day が 1 の行について、Temp 以外の列を抽出
> subset(airquality, Day == 1, select = -Temp)

  Ozone Solar.R Wind Month Day
1     41       190  7.4      5  1
32    NA       286  8.6      6  1
62    135       269  4.1      7  1
93    39        83  6.9      8  1
124   96       167  6.9      9  1
```

(data-subset.r)

その他、データフレームを特定のグループ分けに基づいて分割・結合するための関数 `split()`, `merge()` が用意されている（詳しい使い方はヘルプを参照）。また、より高度なデータフレームの加工を実行するための関数群がパッケージ `dplyr` に用意されている。

演習 3.2. データセット `datasets::airquality` (1973 年 5 月から 9 月までのニューヨークの大気の状態に関するデータ) から以下の条件を満たすデータを取り出せ。

1. 7 月のオゾン濃度 (`Ozone`)
2. 日射量 (`Solar.R`) に欠測 (NA) がないデータの月 (`Month`) と日 (`Day`)
3. 風速 (`Wind`) が時速 10 マイル以上で、気温 (`Temp`) が華氏 80 度以上の日のデータ

3.3 ファイルを用いたデータの読み書き

実際の解析の過程においては、収集されたデータを読み込んだり、整理したデータを保存したりする必要が生じる。R では一般に用いられる CSV 形式 (comma separated values) のテキストファイルと、R の内部表現を用いたバイナリーファイル (ここでは RData 形式と呼ぶ) をサポートしている。以下では、データフレームを対象として、それぞれの形式でファイルの読み書きを行うための関数を纏める。

3.3.1 作業ディレクトリの確認と変更

R の実行は特定のフォルダ（ディレクトリ）上で行われており、そのフォルダを **作業ディレクトリ** と呼ぶ。R のコード内でファイル名を指定した場合、特に指定しない限り作業ディレクトリに存在するものとして扱われる。現在の作業ディレクトリは関数 `getwd()` で確認できる。作業ディレクトリの変更には関数 `setwd()` を利用するか、RStudio 上部の「Session」という項目から「Set Working Directory」を選び、その中の「Choose Directory...」という項目を選択すれば変更する作業ディレクトリを指定することができる。

```
> ### 作業ディレクトリの確認 (環境によって実行結果が異なるため、実行結果は省略)
> getwd()
> ### 作業ディレクトリの移動 (環境によって指定の仕方も異なるので注意)
> setwd("~/Documents") # ホームディレクトリ下の「書類」フォルダに移動
>
```

(data-wd.r)

3.3.2 CSV 形式の操作

1 つのデータフレームを CSV 形式のファイルへ書き出すには関数 `write.csv()` を用いる。書き出し後のファイルは特に指定しない限り作業ディレクトリ下に保存される。

```
> ### 関数 write.csv の使い方
> (mydata <- subset(airquality, Ozone>90, select=-Temp)) # データフレームの作成
```

3 データの加工・整理と入出力

```
Ozone Solar.R Wind Month Day
30     115    223  5.7    5  30
62     135    269  4.1    7  1
69      97    267  6.3    7  8
70      97    272  5.7    7  9
86     108    223  8.0    7  25
99     122    255  4.0    8  7
101    110    207  8.0    8  9
117    168    238  3.4    8  25
121    118    225  2.3    8  29
124     96    167  6.9    9  1
127     91    189  4.6    9  4

> dim(mydata) # 大きさを確認
[1] 11 5
> write.csv(mydata,file="mydata.csv") # csv ファイルとして書き出し
(data-write.csv.r)
```

CSV 形式のファイルから読み込むには関数 `read.csv()` を用いる。読み込むファイルは、ディレクトリを明示的に指定しない限り、作業ディレクトリに保存しておかなくてはいけない。

```
> ### 関数 read.csv の使い方
> (newdata <- read.csv(file="mydata.csv",row.names=1)) # csv ファイルの読み込み

Ozone Solar.R Wind Month Day
30     115    223  5.7    5  30
62     135    269  4.1    7  1
69      97    267  6.3    7  8
70      97    272  5.7    7  9
86     108    223  8.0    7  25
99     122    255  4.0    8  7
101    110    207  8.0    8  9
117    168    238  3.4    8  25
121    118    225  2.3    8  29
124     96    167  6.9    9  1
127     91    189  4.6    9  4

> dim(newdata) # 大きさを確認
[1] 11 5

> ### 東京都の 2016 年の気候データによる例 (kikou2016.csv)
> ### 気象庁のホームページより取得
> ### http://www.data.jma.go.jp/gmd/risk/obsdl/index.php
> ### 東京都の 2016 年の各日の
> ### 平均気温 (°C)・降水量 (mm)・全天日射量 (MJ/u)・平均風速 (m/s)
> kikou <- read.csv("kikou2016.csv",
+                     fileEncoding="sjis") # 文字コードが Shift-JIS のため
> head(kikou)      # データの最初の 6 行を表示

月 日 気温 降水量 日射量 風速
1 1 1 7.5 0 11.80 2.6
2 1 2 7.3 0 11.59 1.9
3 1 3 9.3 0 10.77 1.4
4 1 4 9.2 0 11.19 1.6
```

3 データの加工・整理と入出力

```
5 1 5 10.9      0 10.57 1.8
6 1 6 8.9       0 4.54 1.9
> dim(kikou)    # 大きさを確認
[1] 366   6
> colnames(kikou) # 列名を確認
[1] "月"     "日"     "気温"   "降水量" "日射量" "風速"
(data-read.csv.r)
```

オプションとして与えられている `row.names=1` は、第 1 列を読み込んだデータフレームの各行の名前に割り当てる意味している。

3.3.3 RData 形式の操作

RData 形式のファイルへの書き出しは関数 `save()` を用いる。関数 `write.csv()` と同様に、書き出し後のファイルは特に指定しない限り作業ディレクトリ下に保存される。CSV 形式と異なり、複数のデータフレームを 1 つのファイルに同時に保存することもできる。

```
> ### 関数 save の使い方
> (mydat1 <- subset(airquality, Temp>95, select=-Ozone)) # データフレームの作成
  Solar.R Wind Temp Month Day
120     203  9.7   97     8   28
122     237  6.3   96     8   30
> (mydat2 <- subset(airquality, Temp<60, select=-Ozone)) # データフレームの作成
  Solar.R Wind Temp Month Day
5        NA 14.3   56     5   5
8        99 13.8   59     5   8
15       65 13.2   58     5  15
18       78 18.4   57     5  18
21        8  9.7   59     5  21
25       66 16.6   57     5  25
26      266 14.9   58     5  26
27        NA  8.0   57     5  27
> dim(mydat1) # 大きさを確認
[1] 2 5
> dim(mydat2) # 大きさを確認
[1] 8 5
> save(mydat1,mydat2,file="mydata.rdata") # RData 形式で書き出し
(data-save.r)
```

RData 形式のファイルからの読み込みは関数 `load()` を用いる。関数 `read.csv()` と同様に、読み込むファイルはディレクトリを明示的に指定しない限り、作業ディレクトリに置かなくてはならない。

```
> ### 関数 load の使い方
> (mydat1 <- subset(airquality, Ozone > 120)) # データフレームの作成

  Ozone Solar.R Wind Temp Month Day
62     135      269  4.1    84     7   1
99     122      255  4.0    89     8   7
117    168      238  3.4    81     8  25

> load(file="mydata.rdata") # RData 形式の読み込み
> mydat1 # save したときの名前で読み込まれ上書きされる

  Solar.R Wind Temp Month Day
120     203  9.7   97     8  28
122     237  6.3   96     8  30

> mydat2

  Solar.R Wind Temp Month Day
5       NA 14.3   56     5   5
8       99 13.8   59     5   8
15      65 13.2   58     5  15
18      78 18.4   57     5  18
21       8  9.7   59     5  21
25      66 16.6   57     5  25
26     266 14.9   58     5  26
27       NA  8.0   57     5  27
```

(data-load.r)

関数 `save()` では、データフレームの名前と内容が保存されるので、保存された名前が自動的に用いられる。したがって読み込む際には変数名の重複に注意が必要である。

演習 3.3. ファイル操作に慣れよう。

1. 関数 `write.csv()` で書き出したファイルの中身を適当なアプリケーションで確認しなさい。
2. 適当に作成したデータフレームをファイルに書き出しなさい。
3. 表を整理するには Excel などの表計算ソフトを用いるのが簡便であり、多くの表計算ソフトは CSV 形式でもデータが保存できるようになっている。自身の利用するソフトにおいて CSV 形式で保存する方法を調べなさい。
4. Excel 形式のファイルを直接読み込むパッケージもいくつかある。どのようなパッケージがあるか調べなさい。

3.4 データの整理

与えられたデータの総和や平均、最大値・最小値を求めたい状況は頻繁にある。R にはこれらの操作を簡便に実行するための関数としてそれぞれ `sum()`, `mean()`, `max()`, `min()` が用意されている。

```
> ### データの集計
> sum(1:100) # 1 から 100 までの整数の総和
```

3 データの加工・整理と入出力

```
[1] 5050  
> ### 気候データによる例  
> kikou <- read.csv("kikou2016.csv", fileEncoding = "sjis")  
> kion <- kikou$気温 # 気温を取り出す  
> mean(kion) # 平均気温の計算  
  
[1] 16.47022  
  
> max(kion) # 最大値  
[1] 31.9  
  
> min(kion) # 最小値  
[1] 2.8  
  
(data-summary.r)
```

行列もしくはデータフレームが与えられた際には、列(あるいは行)ごとに平均などの統計量を計算したい状況が頻繁にある。そのような計算に便利な関数として関数 `apply()` がある。関数 `apply()` は基本的に以下のようない書式で利用する:

```
apply(X, MARGIN, FUN)
```

ここで、引数 `X` にデータフレームを指定し、`MARGIN` には行ごとの計算には 1 を、列ごとの計算には 2 を指定する。引数 `FUN` には求めたい統計量を計算するための関数を指定する。

なお、総和や平均の場合には、列・行ごとに計算するための専用の関数が用意されており、それらを利用することができる。

```
> ### 行・列ごとの計算  
> x <- matrix(1:100, 4, 25)  
> sum(x) # x の成分全ての和を計算する (mean 等も同様)  
  
[1] 5050  
  
> rowSums(x) # 行ごとの総和  
[1] 1225 1250 1275 1300  
  
> apply(x, 1, sum) # 上と同じ  
[1] 1225 1250 1275 1300  
  
> ### 気候データによる例  
> kikou <- read.csv("kikou2016.csv", fileEncoding = "sjis")  
> x <- subset(kikou, select=-c(月, 日)) # 月日は除いておく  
> colMeans(x) # 列ごとの平均  
  
    気温    降水量    日射量    風速  
16.470219  4.860656 12.681967  2.785246  
  
> apply(x, 2, max) # 列ごとの最大値  
  
    気温 降水量 日射量 風速  
31.90 106.50 29.87 7.20  
  
> sapply(x, max) # 上と同じ
```

3 データの加工・整理と入出力

```
気温 降水量 日射量 風速
31.90 106.50 29.87 7.20

> apply(x, 2, min) # 列ごとの最小値
気温 降水量 日射量 風速
2.80 0.00 1.11 1.20

> ## 自作関数の適用
> apply(x, 2, function(x){sum(x>mean(x))}) # 列ごとに平均より大きいデータ数を計算
気温 降水量 日射量 風速
188      72     157    172

(data-apply.r)
```

データフレームの各行をいくつかのグループにまとめて、グループごとの統計量を計算したい状況も頻繁に生じる。この場合に便利なのが関数 `aggregate()` である。関数 `aggregate()` は基本的に以下のような書式で利用する：

```
aggregate(X, BY, FUN)
```

ここで、引数 `X` にデータフレームを指定し、`BY` には各行が属するグループを指定するベクトルをリストで与える（複数可）。引数 `FUN` には求めたい統計量を計算するための関数を指定する。なお `X` がベクトルの場合には関数 `tapply()` も利用可能である。

```
> ### 気候データによる例
> kikou <- read.csv("kikou2016.csv", fileEncoding = "sjis")
> ## 月ごとの各変数の平均値を求める
> aggregate(kikou[, -(1:2)], by = list(月=kikou$月), FUN = mean)

月 気温 降水量 日射量 風速
1 1 6.080645 2.741935 9.891290 2.393548
2 2 7.227586 1.965517 11.431034 2.889655
3 3 10.141935 3.322581 13.443226 2.812903
4 4 15.446667 4.000000 14.909667 3.263333
5 5 20.161290 4.435484 19.268065 3.383871
6 6 22.353333 5.816667 14.974000 2.926667
7 7 25.374194 2.629032 15.326129 2.674194
8 8 27.116129 13.354839 15.801935 3.096774
9 9 24.400000 9.566667 10.021000 2.436667
10 10 18.722581 3.112903 9.597742 2.441935
11 11 11.406667 4.633333 8.243000 2.466667
12 12 8.864516 2.709677 9.112581 2.641935

> ## 以下のコードも同じ結果を返す
> aggregate(. ~ 月, data = subset(kikou, select = -日), FUN = mean)

月 気温 降水量 日射量 風速
1 1 6.080645 2.741935 9.891290 2.393548
2 2 7.227586 1.965517 11.431034 2.889655
3 3 10.141935 3.322581 13.443226 2.812903
4 4 15.446667 4.000000 14.909667 3.263333
5 5 20.161290 4.435484 19.268065 3.383871
6 6 22.353333 5.816667 14.974000 2.926667
7 7 25.374194 2.629032 15.326129 2.674194
8 8 27.116129 13.354839 15.801935 3.096774
```

```

9   9 24.400000 9.566667 10.021000 2.436667
10 10 18.722581 3.112903 9.597742 2.441935
11 11 11.406667 4.633333 8.243000 2.466667
12 12 8.864516 2.709677 9.112581 2.641935

> ## 月および降水の有無でグループ分け
> aggregate(kikou[, -(1:2)], FUN = mean,
+             by = list(月 = kikou$月, 降水の有無 = (kikou$降水量 > 0)))

```

	月	降水の有無	気温	降水量	日射量	風速
1	1	FALSE	6.340741	0.000000	10.809259	2.351852
2	2	FALSE	6.747619	0.000000	12.790000	2.804762
3	3	FALSE	9.781818	0.000000	15.214091	2.654545
4	4	FALSE	16.041176	0.000000	20.606471	3.400000
5	5	FALSE	20.495652	0.000000	22.570870	3.260870
6	6	FALSE	22.550000	0.000000	21.742143	3.328571
7	7	FALSE	25.838095	0.000000	17.782857	2.700000
8	8	FALSE	28.000000	0.000000	20.305000	3.068750
9	9	FALSE	25.662500	0.000000	13.544375	2.462500
10	10	FALSE	18.761905	0.000000	11.958095	2.461905
11	11	FALSE	11.829412	0.000000	10.241176	2.411765
12	12	FALSE	7.969565	0.000000	10.114348	2.626087
13	1	TRUE	4.325000	21.250000	3.695000	2.675000
14	2	TRUE	8.487500	7.125000	7.863750	3.112500
15	3	TRUE	11.022222	11.444444	9.114444	3.200000
16	4	TRUE	14.669231	9.230769	7.460000	3.084615
17	5	TRUE	19.200000	17.187500	9.772500	3.737500
18	6	TRUE	22.181250	10.906250	9.051875	2.575000
19	7	TRUE	24.400000	8.150000	10.167000	2.620000
20	8	TRUE	26.173333	27.600000	10.998667	3.126667
21	9	TRUE	22.957143	20.500000	5.994286	2.407143
22	10	TRUE	18.640000	9.650000	4.641000	2.400000
23	11	TRUE	10.853846	10.692308	5.630000	2.538462
24	12	TRUE	11.437500	10.500000	6.232500	2.687500

(data-aggregate.r)

演習 3.4. R に含まれるデータセット `datasets::airquality` を用いてデータの整理をしてみよう。

1. 月日以外の変数ごとに平均、最大値および最小値を求めよ。
2. 月ごとの平均、最大値および最小値を求めよ。

3.5 補遺

3.5.1 参考文献

この章に関連する参考書としては以下を挙げておく。

- [1] 藤澤洋徳. **確率と統計**. 東京: 朝倉書店, 2006.
- [2] 吉田朋広. **数理統計学**. 東京: 朝倉書店, 2006.
- [3] 竹内啓. **数理統計学**. 東京: 東洋経済, 1963.
- [4] 金明哲. **Rによるデータサイエンス(第2版)**. 東京: 森北出版, 2017.
- [5] U. リゲス(石田基広訳). **Rの基礎とプログラミング技法**. 東京: 丸善出版, 2012.

- [6] 奥村晴彦. **Rで楽しむ統計**. 東京: 共立出版, 2016.
- [7] Larry Wasserman. **All of Statistics**. New York: Springer, 2004.
- [8] Gareth James et al. **An Introduction to Statistical Learning with Applications in R**. New York: Springer, 2013.
- [9] 山本義郎, 藤野友和, 久保田貴文. **Rによるデータマイニング入門**. 東京: オーム社, 2015.

3.5.2 パッケージ dplyr によるデータの操作

先に紹介した関数 `subset()` を含め, R の基本パッケージにはデータを整理するための簡単な機能を提供する関数が用意されているので, これらの関数を組み合わせて希望の操作を行うことが可能である. しかしながら, 複雑な操作を行う場合にはプログラムが繁雑となるので, より強力な関数群を集めたパッケージがいくつか用意されている. その1つとしてパッケージ `dplyr` がある. 以下では, その基本的な使い方を紹介する.

まず, 以下のようにしてパッケージ `dplyr` を読み込んでおく.

```
> require(dplyr) # パッケージ dplyr を読み込む  
(dplyr-require.r)
```

3.5.3 条件を指定した行の選択

特定の条件を満たす行を選択するには関数 `dplyr::filter()` を用いる. 単一行の番号を指定して選択するには関数 `dplyr::slice()` を用いればよい.

```
> #### dplyr::filter (条件で絞り込む)  
> filter(airquality, Temp>80, Ozone>90) # 条件の and  
  
Ozone Solar.R Wind Temp Month Day  
1 135 269 4.1 84 7 1  
2 97 267 6.3 92 7 8  
3 97 272 5.7 92 7 9  
4 108 223 8.0 85 7 25  
5 122 255 4.0 89 8 7  
6 110 207 8.0 90 8 9  
7 168 238 3.4 81 8 25  
8 118 225 2.3 94 8 29  
9 96 167 6.9 91 9 1  
10 91 189 4.6 93 9 4  
  
> filter(airquality, Day==5 | Day==10) # 条件の or  
  
Ozone Solar.R Wind Temp Month Day  
1 NA NA 14.3 56 5 5  
2 NA 194 8.6 69 5 10  
3 NA 220 8.6 85 6 5  
4 39 323 11.5 87 6 10  
5 64 175 4.6 83 7 5  
6 85 175 7.4 89 7 10  
7 35 NA 7.4 85 8 5  
8 NA 222 8.6 92 8 10  
9 47 95 7.4 87 9 5  
10 24 259 9.7 73 9 10
```

```
> ### dplyr::slice (行番号で絞り込む)
> slice(airquality, seq(2, 16, by=2))
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	36	118	8.0	72	5	2
2	18	313	11.5	62	5	4
3	28	NA	14.9	66	5	6
4	19	99	13.8	59	5	8
5	NA	194	8.6	69	5	10
6	16	256	9.7	69	5	12
7	14	274	10.9	68	5	14
8	14	334	11.5	64	5	16

(dplyr-filter.r)

3.5.4 列の値による行の並べ替え

特定の列に入っている値の昇順、あるいは降順に並べ替えるには関数 `dplyr::arrange()` を用いる。特に指定がなければ昇順で並べ替えが行われるが、降順で並べ替えたい場合には関数 `dplyr::desc()` の中に入れて列を指定すればよい。

```
> ### dplyr::arrange (指定した変数の順に並べ替える)
> arrange(airquality, Temp, Wind) %>%head(12) # 基本は昇順
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	NA	NA	14.3	56	5	5
2	NA	NA	8.0	57	5	27
3	NA	66	16.6	57	5	25
4	6	78	18.4	57	5	18
5	18	65	13.2	58	5	15
6	NA	266	14.9	58	5	26
7	1	8	9.7	59	5	21
8	19	99	13.8	59	5	8
9	4	25	9.7	61	5	23
10	32	92	12.0	61	5	24
11	8	19	20.1	61	5	9
12	11	44	9.7	62	5	20


```
> arrange(airquality, Temp, desc(Wind)) %>%head(12) # Wind は降順
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	NA	NA	14.3	56	5	5
2	6	78	18.4	57	5	18
3	NA	66	16.6	57	5	25
4	NA	NA	8.0	57	5	27
5	NA	266	14.9	58	5	26
6	18	65	13.2	58	5	15
7	19	99	13.8	59	5	8
8	1	8	9.7	59	5	21
9	8	19	20.1	61	5	9
10	32	92	12.0	61	5	24
11	4	25	9.7	61	5	23
12	18	313	11.5	62	5	4


```
> ## %>% は処理を渡す演算子。head(12) で出力結果を 12 行に制限する
```

(dplyr-arrange.r)

3.5.5 列の選択

特定の列または列の集合を選択するには関数 `dplyr::select()` を用いる。列の指定に仕方には、必要とする列を指定する方法と、不要な列を除く方法がある。また、選択と同時に列名を変更することもできる。なお、選択せずに名前のみを変更するには関数 `dplyr::rename()` を用いる。

```
> ### dplyr::select (列を選択する)
> select(airquality, Temp, Wind, Ozone) %>%head(12) # 列記する場合

  Temp Wind Ozone
1    67   7.4    41
2    72   8.0    36
3    74  12.6    12
4    62  11.5    18
5    56  14.3    NA
6    66  14.9    28
7    65   8.6    23
8    59  13.8    19
9    61  20.1     8
10   69   8.6    NA
11   74   6.9     7
12   69   9.7    16

> select(airquality, Ozone:Temp) %>%head(12) # 最初と最後の列を指定する場合

  Ozone Solar.R Wind Temp
1      41       190   7.4   67
2      36       118   8.0   72
3      12       149  12.6   74
4      18       313  11.5   62
5      NA       NA  14.3   56
6      28       NA  14.9   66
7      23       299   8.6   65
8      19       99  13.8   59
9      8        19  20.1   61
10     NA       194   8.6   69
11     7        NA   6.9   74
12     16       256   9.7   69

> select(airquality, -(Month:Day)) %>%head(12) # 削除する場合

  Ozone Solar.R Wind Temp
1      41       190   7.4   67
2      36       118   8.0   72
3      12       149  12.6   74
4      18       313  11.5   62
5      NA       NA  14.3   56
6      28       NA  14.9   66
7      23       299   8.6   65
8      19       99  13.8   59
9      8        19  20.1   61
10     NA       194   8.6   69
11     7        NA   6.9   74
12     16       256   9.7   69

> select(airquality, TempInF=Temp) %>%head(12) # 名前を変更する場合

  TempInF
1       67
```

3 データの加工・整理と入出力

```
2      72
3      74
4      62
5      56
6      66
7      65
8      59
9      61
10     69
11     74
12     69
```

> `rename(airquality, TempInF=Temp) %>%head(12)` # 指定箇所だけ名前を変更する場合

```
Ozone Solar.R Wind TempInF Month Day
1     41    190  7.4     67      5   1
2     36    118  8.0     72      5   2
3     12    149 12.6     74      5   3
4     18    313 11.5     62      5   4
5     NA     NA 14.3     56      5   5
6     28     NA 14.9     66      5   6
7     23    299  8.6     65      5   7
8     19     99 13.8     59      5   8
9      8     19 20.1     61      5   9
10    NA    194  8.6     69      5  10
11     7     NA  6.9     74      5  11
12    16    256  9.7     69      5  12
```

(dplyr-select.r)

3.5.6 値の整理

各列でどのような値が使われているかを知るには関数 `dplyr::distinct()` を用いる。こうした集計は特にカテゴリカル変数の場合に重要となり、複数の列での組み合わせにも対応している。

```
> ### dplyr::distinct (異なる値を取り出す)
> distinct(airquality, Month)          # どの月を対象としているか調べる

Month
1     5
2     6
3     7
4     8
5     9

> distinct(airquality, Ozone, Temp) %>%head(12) # Ozone と Temp の異なる組み合わせ

Ozone Temp
1     41   67
2     36   72
3     12   74
4     18   62
5     NA   56
6     28   66
7     23   65
8     19   59
```

```

9      8   61
10     NA  69
11      7  74
12     16  69

> #### dplyr::count (異なる値がいくつあるか数える)
> count(airquality, Month)          # 各月のデータ数を調べる

# A tibble: 5 x 2
  Month     n
  <int> <int>
1     5     31
2     6     30
3     7     31
4     8     31
5     9     30

(dplyr-distinct.r)

```

3.5.7 列の追加

新しい量を計算して、新たな列を加えるには関数 `dplyr::mutate()` を用いる。新たな列のみで他の列を保持する必要がない場合には関数 `dplyr::transmute()` を用いればよい。

```

> #### dplyr::mutate (新しい列を作成する)
> mutate(airquality, TempC=(Temp-32)/1.8) %>%head(12) # 華氏を摂氏に変換

  Ozone Solar.R Wind Temp Month Day    TempC
1     41     190  7.4   67     5   1 19.44444
2     36     118  8.0   72     5   2 22.22222
3     12     149 12.6   74     5   3 23.33333
4     18     313 11.5   62     5   4 16.66667
5     NA     NA 14.3   56     5   5 13.33333
6     28     NA 14.9   66     5   6 18.88889
7     23     299  8.6   65     5   7 18.33333
8     19      99 13.8   59     5   8 15.00000
9      8      19 20.1   61     5   9 16.11111
10    NA     194  8.6   69     5  10 20.55556
11     7     NA  6.9   74     5  11 23.33333
12    16     256  9.7   69     5  12 20.55556

> transmute(airquality, TempF=Temp, TempC=(Temp-32)/1.8) %>%head(12) # 保持しない

  TempF    TempC
1     67 19.44444
2     72 22.22222
3     74 23.33333
4     62 16.66667
5     56 13.33333
6     66 18.88889
7     65 18.33333
8     59 15.00000
9     61 16.11111
10    69 20.55556
11    74 23.33333
12    69 20.55556

(dplyr-mutate.r)

```

3.5.8 データフレームの集計

データフレームの各行での集計値を得るには関数 `dplyr::summarize()` を用いる。どのような集計を行うかは適切な関数を指定する必要がある。

```
> ### dplyr::summarize (データフレームを集計する)
> summarize(airquality,
+           mean      = mean(Ozone, na.rm=TRUE),
+           min       = min(Ozone, na.rm=TRUE),
+           median   = median(Ozone, na.rm=TRUE),
+           max       = max(Ozone, na.rm=TRUE))

  mean min median max
1 42.12931    1    31.5 168

(dplyr-summarize.r)
```

3.5.9 行のリサンプリング

データフレームの中からランダムに行をリサンプリングするには関数 `dplyr::sample_n()` または `dplyr::sample_frac()` を用いる。関数 `dplyr::sample_n()` は指定した個数で、関数 `dplyr::sample_frac()` は指定して割合で全体の中からリサンプリングを行う。これらはブートストラップ法 (bootstrap method) や交差検証法 (cross-validation method) などに利用される。

```
> ### dplyr::sample_n (個数を指定してサンプリングする)
> sample_n(airquality, size=10)

  Ozone Solar.R Wind Temp Month Day
1     85      175  7.4   89      7   10
2     NA      258  9.7   81      7   22
3      9      36 14.3   72      8   22
4     44      236 14.9   81      9   11
5     32      92 15.5   84      9    6
6    135      269  4.1   84      7    1
7     14      20 16.6   63      9   25
8     11      290  9.2   66      5   13
9     59      51  6.3   79      8   17
10    47      95  7.4   87      9    5

> ### dplyr::sample_frac (比率を指定してサンプリングする)
> sample_frac(airquality, size=0.05)

  Ozone Solar.R Wind Temp Month Day
1     NA      186  9.2   84      6    4
2     23      115  7.4   76      8   18
3     64      253  7.4   83      7   30
4     35      NA  7.4   85      8    5
5     36      139 10.3   81      9   23
6     13      27 10.3   76      9   18
7     21      259 15.5   76      9   12
8      9      36 14.3   72      8   22

(dplyr-sample.r)
```

なお、復元抽出を行う場合には関数 `sample()` と同様にオプション `replace=TRUE` を付ければよい。

3.5.10 データフレームのグループ化

特定の行でグループ化して、その集計を求めるには関数 `dplyr::group_by()` を用いる。この関数は、上記の関数と組み合わせ、集計途中を保存しながら次の関数に引き渡すようにして用いることが多い。順次操作を行う操作を簡略化するために演算子 `%>%` が用意されている。

```
> ### dplyr::group_by (指定した変数でグループ化する)
> air_month <- group_by(airquality, Month) # 月ごとにグループ化
> summarize(air_month,
+            count    = n(), # 各月の日数を求める
+            mean     = mean(Ozone, na.rm=TRUE),
+            min      = min(Ozone, na.rm=TRUE),
+            median   = median(Ozone, na.rm=TRUE),
+            max      = max(Ozone, na.rm=TRUE))

# A tibble: 5 x 6
  Month count  mean  min median  max
  <int> <int> <dbl> <dbl>  <dbl> <dbl>
1     5     31  23.6    1     18   115
2     6     30  29.4   12     23    71
3     7     31  59.1    7     60   135
4     8     31  60.0    9     52   168
5     9     30  31.4    7     23    96

> ##同じことは演算子 %>% を用いることで簡略化できる
> airquality %>%
+   group_by(Month) %>%
+   summarize(
+     count    = n(),
+     mean     = mean(Ozone, na.rm=TRUE),
+     min      = min(Ozone, na.rm=TRUE),
+     median   = median(Ozone, na.rm=TRUE),
+     max      = max(Ozone, na.rm=TRUE))

# A tibble: 5 x 6
  Month count  mean  min median  max
  <int> <int> <dbl> <dbl>  <dbl> <dbl>
1     5     31  23.6    1     18   115
2     6     30  29.4   12     23    71
3     7     31  59.1    7     60   135
4     8     31  60.0    9     52   168
5     9     30  31.4    7     23    96
```

(dplyr-groupby.r)

演習 3.5. データフレームを整理してみよう。

1. 関数 `data()` で調べた適当なデータフレームを整理しなさい。

2. 開発者たちによる解説は以下のサイトにある。

<https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html>

パッケージ `nycflights13` をインストールして確認しなさい。

3.5.11 その他

CSV 形式でない通常のテキストファイルを読み込むための関数として `read.table()` がある。ファイルの大きさが大きい場合、読み込みや書き出しに非常に時間がかかることがある。その場合、ファイル操作を高速化したパッケージ群がいくつか開発されているため、それらを利用するのが便利である。例えば、大規模データの読み込みにはパッケージ `data.table` の関数 `fread()` が便利である。CSV ファイルの書き出しにはパッケージ `readr` の関数 `write_csv()` が便利である。

4 データのプロット

記述統計量と並んでデータ全体の特徴や傾向を把握するために効果的な方法は、データを可視化することである。R の基本パッケージ `graphics` に用意されている作図機能はきわめて多彩であり、これらを適切に組み合わせることによって様々な種類のグラフを描くことができる。以下では、いくつかの代表的な描画関数を取り上げて解説する。

描画関連の関数は色、線種や線の太さ、あるいは図中の文字の大きさなどを指定するために、多彩なオプションを用意しているので、必要に応じて関数 `help()`(ヘルプの表示)と `example()`(例題の表示)を利用して欲しい。

4.1 基本的な描画

描画において基本となるのは関数 `plot()` である。

関数 `sin()` のように 1 変数の関数として定義されているものは、定義域を指定してやればそのまま表示することができる。関数を追加するにはオプション `add` とともに関数 `curve()` を用いれば良い。

また、関数 `plot()` に同じ長さの二つのベクトルを与えると、同じ番号の要素からなる点の組 (x, y) をプロットして、その**散布図**を描くことができる。

プロットの種類(点や線)を指定するにはオプション `type` を用いる。`'p'` で点(point), `'l'` で点列を順に結んだ線(line)が描かれる。なお、オプションに与える文字列は'(シングルクオート)か"(ダブルクオート)で囲む必要がある。

オプション `col` で”色の名前”を指定することにより点や線の色を変えることができる。R で指定することのできる色の名前は関数 `colors()` で照会することができる。

関数 `plot()` で描いた図中に更に線を追加するには関数 `lines()` を、点を追加するには関数 `points()` を用いる。

これ以外にも関数 `plot()` は様々なオプションを指定することができるので、`help(plot)` および `help(plot.default)` を参照して欲しい。

[Figure 4.1 を参照]

```
> ### ベクトルの描画
> plot(11:20)
> ### 関数の描画
> plot(sin, 0, 4*pi,
+       col="blue", # グラフの線の色
+       lwd=2, # グラフの線の太さ
+       ylab="sin/cos" # y 軸のラベル
+ )
> curve(cos,
+        add=TRUE, # グラフを上書き
+        col="red", lwd=2)
> x <- seq(0, 4*pi, by=0.1)
> y <- sin(x) + rep_len(c(-0.2, 0.1), length(x))
> points(x, y, col="green", pch="*") # 点を追加。pch は点の形を指定
> ### データ点の描画
> plot(x, y, type="p", pch="x", ylim=c(-2,2)) # ylim で値域を指定
> curve(sin, add=TRUE, col="orange", lwd=2)
```

4 データのプロット

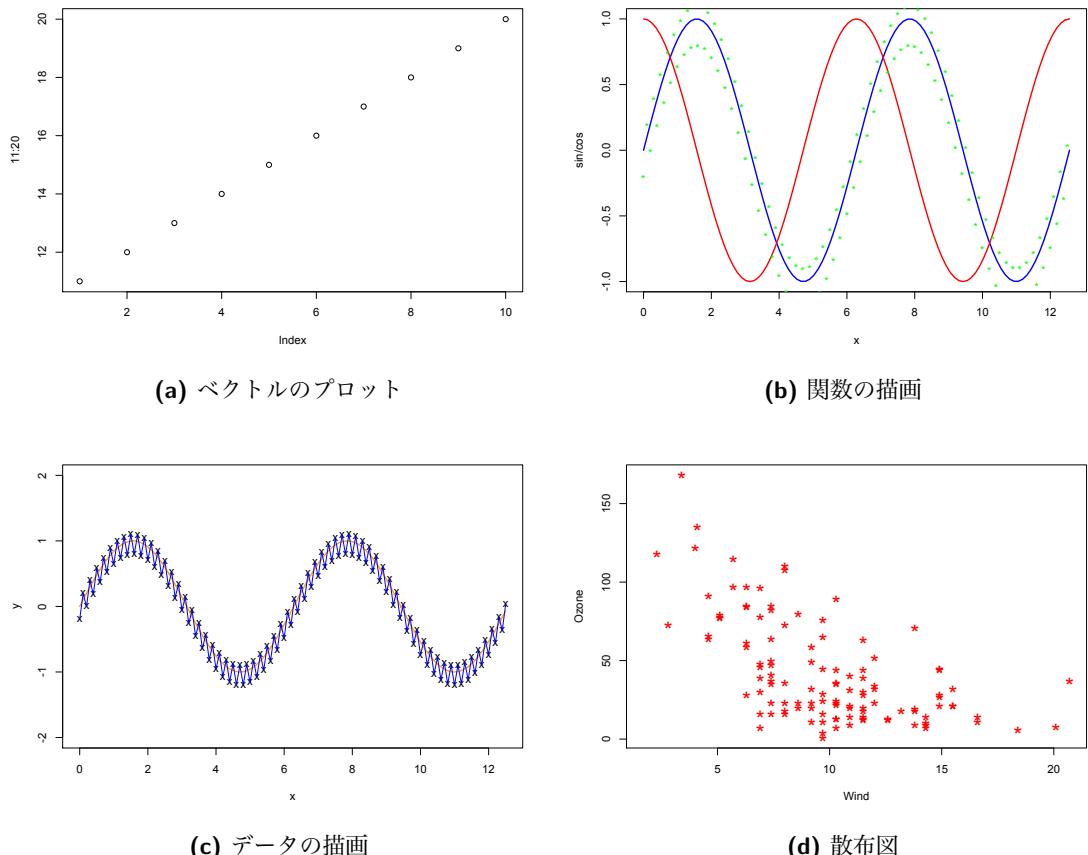


Figure 4.1: 関数 `plot()` の例

```
> lines(x, y, col="blue") # 折れ線を追加
> ### データフレームを用いた散布図 (airquality を利用)
> plot(Ozone ~ Wind, data=airquality,
+       pch="*", col="red", cex=2) # cex は点の大きさの倍率を指定
```

(graph-plot.r)

関数 `legend()` によってグラフに凡例を追加することができる。また、以下の例で見るよう *R* には数式を扱う機能がある。詳細は `help(plotmath)` を参照してほしい。

```
> ### 凡例の追加
> f <- function(x) exp(-x) * cos(x)
> plot(f, 0, 2*pi, col="red", lwd=2, ylab="")
> g <- function(x) exp(-x) * sin(x)
> curve(g, lty=2, # グラフの線の形式 2 は破線
+        add=TRUE, col="blue", lwd=2)
> legend(4, # 凡例の左上の x 座標
+        1, # 凡例の左上の y 座標
+        legend=c(expression(e^{-x} * cos(x)), expression(e^{-x} * sin(x))), # 凡例
+        lty=c(1,2), lwd=2, col=c("red", "blue"), # パラメータはグラフに準拠
+        bty="n", # 凡例の枠線の形式 (オプション) "n" は枠線なし
+        y.intersp=2) # 行間の指定 (オプション)
```

4 データのプロット

(graph-legend.r)

なお、OSによっては日本語を含む図を描画すると文字化けする場合がある。その場合関数 `par()` のオプション `family` に適当なフォントファミリーを指定することで文字化けを回避できる場合がある。例えば Mac OS のデフォルトの設定では日本語を含む図は文字化けしてしまうが、以下のコマンドをコンソール上で実行することで文字化けを回避できる。

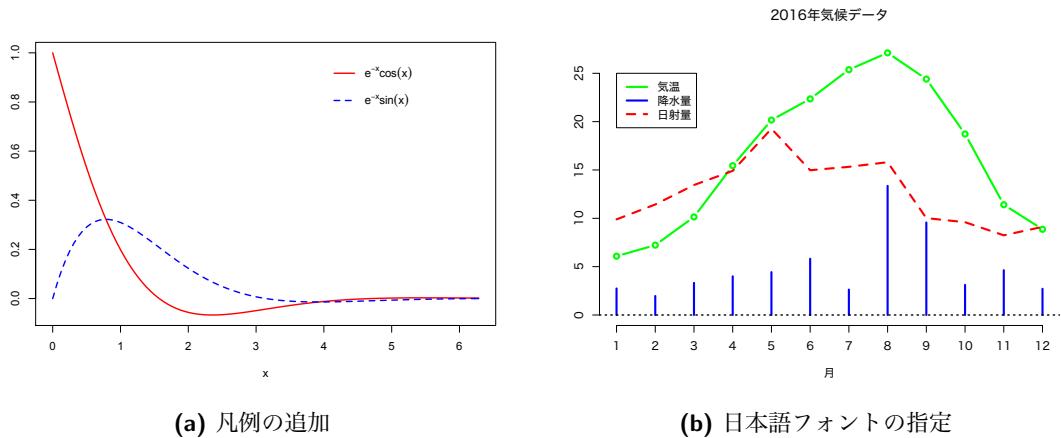


Figure 4.2: より進んだ描画の例

[Figure 4.2 を参照]

```
> ### 日本語フォントの指定
> par(family = "HiraginoSans-W4")
> ### 東京都の2016年の気候データによる例
> ### 気象庁のホームページより取得
> ### http://www.data.jma.go.jp/gmd/risk/obsdl/index.php
> ### 東京都の2016年の各日の平均気温(℃)・降水量(mm)・全天日射量(MJ/u)・
> ### 平均風速(m/s)を記録したデータセット kikou2016.csv
> kikou <- read.csv("kikou2016.csv", fileEncoding = "sjis")
> ## 月ごとの平均をプロットする
> (x <- aggregate(kikou[, -c(1, 2)], by = list(月 = kikou$月),
+ FUN = "mean")) # 月ごとの平均を計算
```

	月	気温	降水量	日射量	風速
1	1	6.080645	2.741935	9.891290	2.393548
2	2	7.227586	1.965517	11.431034	2.889655
3	3	10.141935	3.322581	13.443226	2.812903
4	4	15.446667	4.000000	14.909667	3.263333
5	5	20.161290	4.435484	19.268065	3.383871
6	6	22.353333	5.816667	14.974000	2.926667
7	7	25.374194	2.629032	15.326129	2.674194
8	8	27.116129	13.354839	15.801935	3.096774
9	9	24.400000	9.566667	10.021000	2.436667
10	10	18.722581	3.112903	9.597742	2.441935
11	11	11.406667	4.633333	8.243000	2.466667
12	12	8.864516	2.709677	9.112581	2.641935

```
> plot(x$気温, type = "b", lwd=3, col="green", ylim=c(0, max(x$気温)),
+       xlab="月", ylab="", main="2016年気候データ", # グラフタイトル
+       axes=FALSE) # 軸を書かない
```

4 データのプロット

```
> axis(1, 1:12, 1:12) # x 軸の作成
> axis(2) # y 軸の作成
> lines(x$降水量, type="h", lwd=3, col="blue")
> lines(x$日射量, lwd=3, lty=2, col="red")
> abline(0, 0, lwd=2, lty="dotted") # y=0 の線を引く
> legend(1, 25, legend=c("気温", "降水量", "日射量"),
+         col=c("green", "blue", "red"), lwd=3,
+         lty=c(1, 1, 2))
```

(graph-font.r)

作成したグラフは保存することができる。RStudio の機能を使う場合には右下ペインの “Plots” タブの “Export” をクリックすると、形式やサイズを指定して保存できる（もしくはクリップボードにコピーもできる）。コマンドで実行することも可能であるが、それについての詳細は `help(png)` や `help(dev.copy)` を参照してほしい。

4.2 ヒストグラム

データの頻度分布を表すヒストグラムを描画するには関数 `hist()` を用いる。これ以外にも凝ったヒストグラムを書くための関数がいくつか用意されているが、これらについては `help.search("histogram")` を参照して欲しい。

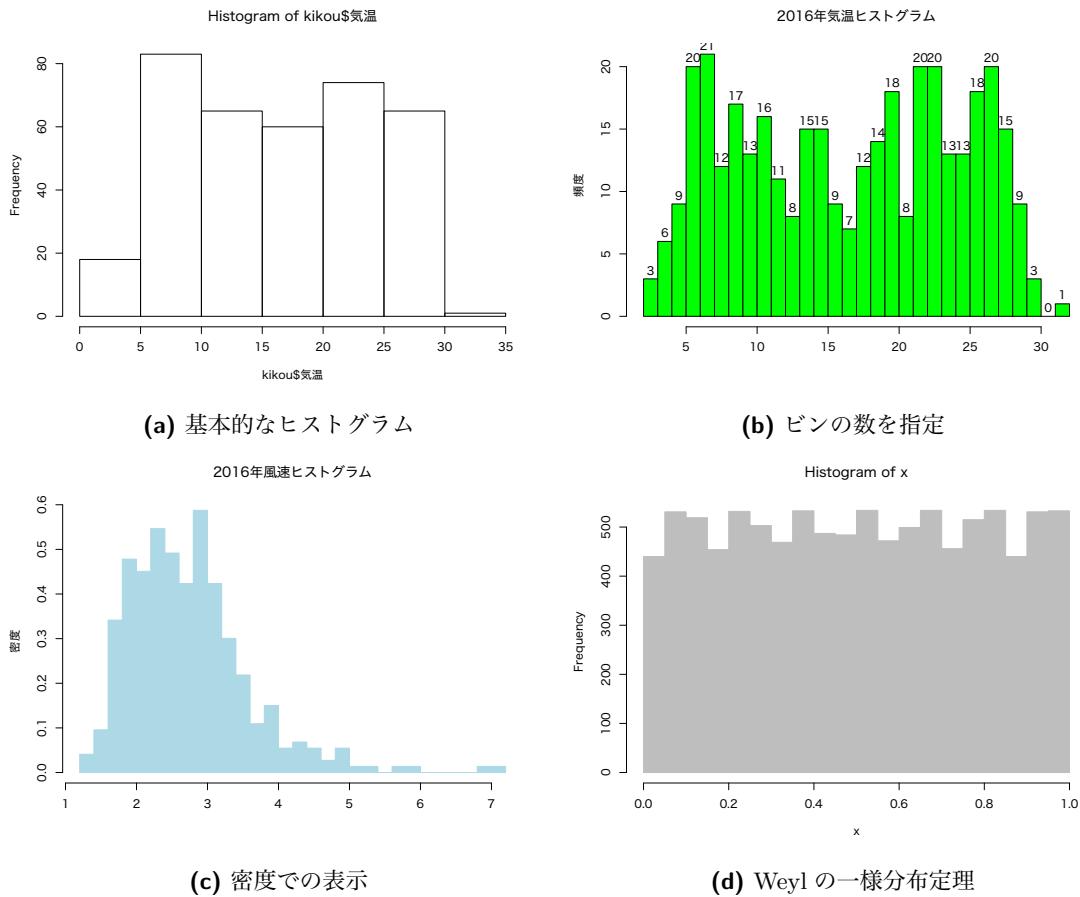


Figure 4.3: 関数 `hist()` の例

[Figure 4.3 を参照]

```
> ### 関数 hist によるヒストグラムの作図
> kikou <- read.csv("kikou2016.csv", fileEncoding = "sjis")
> ### 基本的なヒストグラム
> hist(kikou$気温)
> ### ビンの数を指定
> hist(kikou$気温,
+       xlab="", ylab="頻度",
+       breaks=25, # ビンの数を約 25 に設定
+       labels=TRUE, # 各ビンの度数を表示
+       col="green", main="2016 年気温ヒストグラム")
> ### 密度での表示
> hist(kikou$風速, freq = FALSE, # 全体に対する割合で表示
+       xlab="", ylab="密度", breaks=25, col="lightblue",
+       border="lightblue", # 長方形の境界の色
+       main="2016 年風速ヒストグラム")
> ### Weyl の一様分布定理の確認
> ### a が無理数のとき, 数列 a, 2a, 3a, ... の小数部分は区間 (0, 1) 上に均一に現れる
> a <- pi # 無理数
> n <- 10000
> x <- (1:n) * a
> x <- x - floor(x) # 小数部分の計算 (floor はいわゆる Gauss 記号)
> hist(x, breaks=20, col="gray", border="gray")
```

(graph-hist.r)

4.3 箱ひげ図

複数のデータの分布を比較する際、観測数が大きく異なるなどヒストグラムでの比較が難しい場合がある。複数のデータの分布の違いを見るには箱ひげ図(boxplot)が良く用いられるが、これは関数 `boxplot()` で描くことができる。

[Figure 4.4 を参照]

```
> ### 関数 boxplot による箱ひげ図の作図
> kikou <- read.csv("kikou2016.csv", fileEncoding="sjis")
> ### 基本的な箱ひげ図
> boxplot(kikou[, -c(1, 2)]) # 月日は除く
> ### 月ごとに気温を分類
> boxplot(気温 ~ 月, data=kikou, col="orange", main="月ごとの気温")
> ### 図を回転
> boxplot(気温 ~ 月, data=kikou,
+           col="purple", main="月ごとの気温", horizontal=TRUE)
```

(graph-boxplot.r)

4.4 棒グラフ

関数 `barplot()` によって棒グラフを作成できる。`barplot()` の第1引数はベクトルまたは行列でなければならないことに注意する。

4 データのプロット

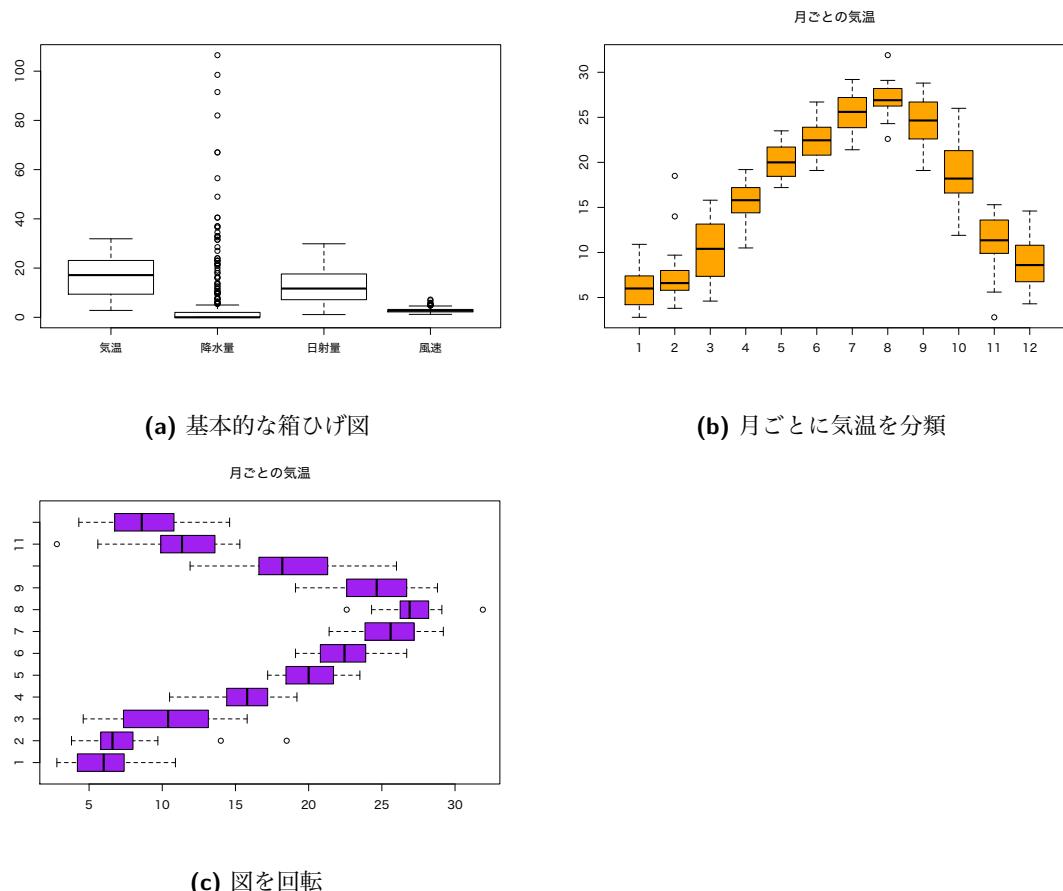


Figure 4.4: 関数 boxplot() の例

[Figure 4.5 を参照]

```
> ### 関数 barplot による棒グラフの作図
> kikou <- read.csv("kikou2016.csv", fileEncoding="sjis")
> ## 月ごとに各変数の平均を計算
> x <- aggregate(kikou[, -c(1, 2)], by=list(月 = kikou$月),
+                  FUN = "mean")
> ### 基本的な棒グラフ
> barplot(x[, 2], # 棒の高さのベクトル
+           col="slateblue", # 棒の色の指定
+           names.arg=x[, 1], # x 軸のラベル
+           main=names(x)[2]) # タイトル
> ### 複数の棒グラフ
> barplot(as.matrix(x[, -1]), # 第 1 引数はベクトルまたは行列
+           col=rainbow(12)[c(8:1, 12:9)], # 12 色に色分け
+           beside=TRUE, # 棒グラフを横に並べる
+           space=c(1.5, 3), # 棒グラフ間・変数間のスペースを指定
+           legend.text=paste0(x[, 1], "月"), # 凡例の指定
+           args.legend=list(ncol = 2)) # 凡例を 2 列にして表示
```

(graph-barplot.r)

4 データのプロット

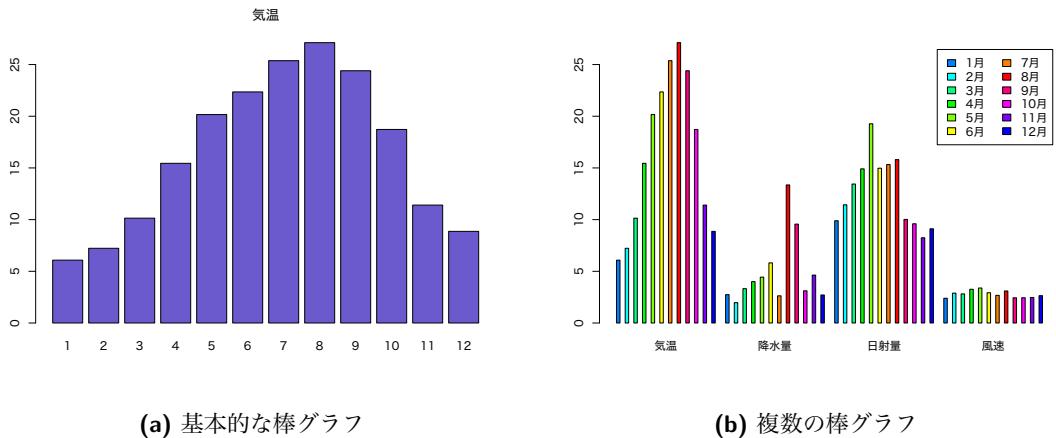


Figure 4.5: 関数 barplot() の例

4.5 円グラフ

円グラフは関数 pie() で描くことができる。

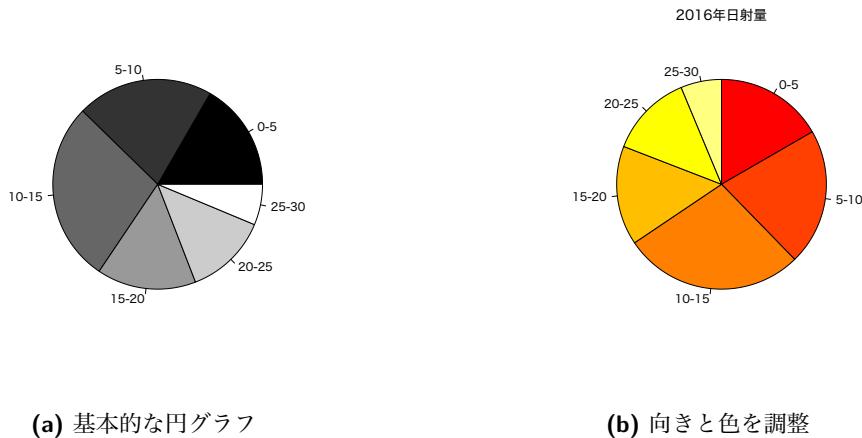


Figure 4.6: 関数 pie() の例

[Figure 4.6 を参照]

```
> ### 関数 pie による円グラフの作図
> kikou <- read.csv("kikou2016.csv", fileEncoding="sjis")
> z <- hist(kikou$日射量, breaks=5, plot=FALSE) # 5つ程度に分類
> x <- z$count
> y <- z$breaks
> names(x) <- paste(y[-length(y)], y[-1], sep="-")
> ### 基本的な円グラフ
> pie(x, col=gray(seq(0,1,length=length(x))))
> ### 向きと色を調整
> pie(x, clockwise=TRUE, col=heat.colors(length(x)), main="2016年日射量")
```

(graph-pie.r)

4.6 散布図行列

多次元データの変数間の関係を概観するために、2つの変数間の散布図を複数行列状に並べた図を用いることがある。これは関数 `pairs()` によって作成することができる（関数 `plot()` でも同じことができる）。

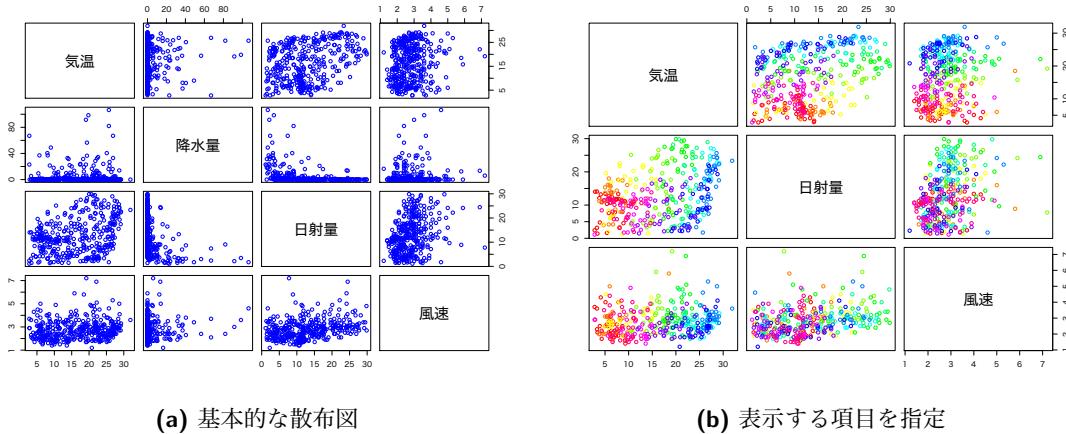


Figure 4.7: 関数 `pairs()` の例

[Figure 4.7 を参照]

```
> ### 関数 pairs による散布図の作図
> kikou <- read.csv("kikou2016.csv", fileEncoding="sjis")
> ### 基本的な散布図
> pairs(kikou[,-c(1,2)], col="blue")
> ## plot(kikou[,-c(1,2)], col="blue") でも同じ図が描ける
> ### 表示する項目を指定
> pairs(~ 気温 + 日射量 + 風速, data = kikou,
+       col=rainbow(12)[kikou$月]) # 月毎に異なる色で表示
```

(graph-pairs.r)

4.7 3次元のグラフ

3次元のグラフを2次元に射影した俯瞰図は、関数 `persp()` を用いて描くことができる。視線の方向はオプション `theta` と `phi` で極座標を指定することによって制御することができる。パッケージ `scatterplot3d` には、3次元の散布図を書くための関数 `scatterplot3d()` が用意されている。

[Figure 4.8 を参照]

```
> ### 関数 persp による2変数関数の俯瞰図
> f <- function(x,y) x^2 - y^2
> x <- seq(-3, 3, length=51) # x 座標の定義域の分割
> y <- seq(-3, 3, length=51) # y 座標の定義域の分割
> z <- outer(x, y, f) # z 座標の計算
> ### 基本的な俯瞰図
> persp(x, y, z, col="lightblue")
```

4 データのプロット

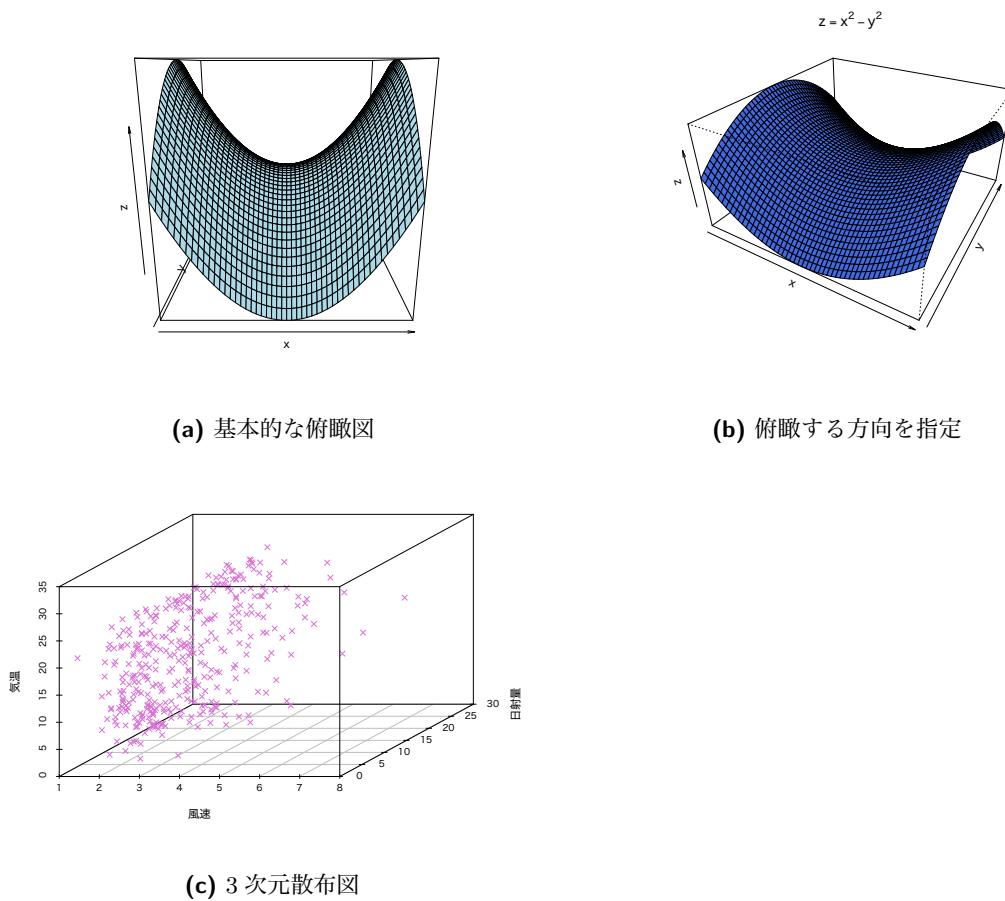


Figure 4.8: 3 次元グラフの例

```
> ### 俯瞰する向きを指定
> persp(x, y, z, theta=30, phi=30, expand=0.5, col="royalblue",
+         main = expression(z==x^2-y^2))
> ### 3次元散布図(パッケージ scatterplot3d を利用)
> ## install.packages("scatterplot3d") # 初めて使う時に必要
> require(scatterplot3d) # パッケージのロード
> kikou <- read.csv("kikou2016.csv", fileEncoding="sjis")
> dat <- subset(kikou, select=c("風速", "日射量", "気温"))
> scatterplot3d(dat, pch=4, color="orchid")
```

(graph-plot3d.r)

4.8 プロット環境の設定

プロットの際の線の種類や色、点の形等のデフォルト値は関数 `par()` で設定できる。設定可能なグラフィックスパラメータは `help(par)` で確認できる。特に以下の例のように関数 `par()` によってプロット環境の設定(複数図の配置、余白の設定など)ができる。

[Figure 4.9 を参照]

```
> ### 複数図の配置
```

4 データのプロット

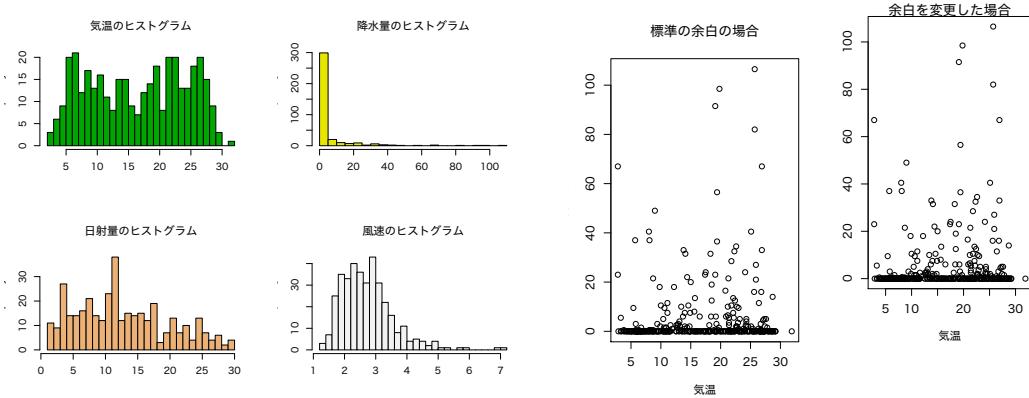


Figure 4.9: 環境の設定の例

```
> ## 気候データの各変数のヒストグラムを 1つの画面に配置
> kikou <- read.csv("kikou2016.csv", fileEncoding="sjis")
> op <- par(mfrow=c(2,2)) # 画面を 2x2 に分割, 行ごとにプロット
> ## par(mfcoll=c(2,2)) で列ごとにプロットできる
> cl <- terrain.colors(4) # 色を用意
> nam <- colnames(kikou)[-c(1:2)] # ヒストグラムを作成する変数名
> ## 第 1 变数のヒストグラムの作成
> hist(kikou[,nam[1]], col=cl[1], breaks=25, xlab="",
+       main=paste0(nam[1], "のヒストグラム"))
> ## 第 2 变数のヒストグラムの作成
> hist(kikou[,nam[2]], col=cl[2], breaks=25, xlab="",
+       main=paste0(nam[2], "のヒストグラム"))
> ## 残りは for 文で作成
> for(i in 3:4){
+   hist(kikou[,nam[i]], col=cl[i], breaks=25, xlab="",
+         main=paste0(nam[i], "のヒストグラム"))
+ }
> par(op) # 設定解除
> ### 余白の設定
> op1 <- par(mfrow=c(1,2))
> plot(kikou[,3:4], main="標準の余白の場合")
> op2 <- par(mar=c(9,2,1,6))
> ## 下・左・上・右の順で余白を設定
> ## デフォルトは par(mar=c(5,4,4,2)+0.1)
> plot(kikou[,3:4], main="余白を変更した場合")
> par(op1); par(op2) # 設定解除
```

(graph-par.r)

演習 4.1. 前章で整理したデータフレームを描画してみよう。

1. 関数 `data()` で調べた適当なデータフレームを描画しなさい。
2. 上記のデータフレームを集計し、その結果を描画しなさい。

4.9 補遺

4.9.1 参考文献

この章に関連する参考書としては以下を挙げておく。

- [1] 藤澤洋徳. **確率と統計**. 東京: 朝倉書店, 2006.
- [2] 吉田朋広. **数理統計学**. 東京: 朝倉書店, 2006.
- [3] 竹内啓. **数理統計学**. 東京: 東洋経済, 1963.
- [4] 金明哲. **Rによるデータサイエンス(第2版)**. 東京: 森北出版, 2017.
- [5] U. リゲス(石田基広訳). **Rの基礎とプログラミング技法**. 東京: 丸善出版, 2012.
- [6] 奥村晴彦. **Rで楽しむ統計**. 東京: 共立出版, 2016.
- [7] Larry Wasserman. **All of Statistics**. New York: Springer, 2004.
- [8] Gareth James et al. **An Introduction to Statistical Learning with Applications in R**. New York: Springer, 2013.
- [9] 山本義郎, 藤野友和, 久保田貴文. **Rによるデータマイニング入門**. 東京: オーム社, 2015.

4.9.2 パッケージ `ggplot` の利用

データの可視化は、データ解析において基本的かつ有効な方法であるのみならず、分析結果を他の人々に説明する際の資料としても必須のものである。そのため R のグラフィック機能を拡張するためのパッケージも多数開発されている。その中でも、近年利用が広まっているものにパッケージ `ggplot2` がある。`ggplot2` は、統一的な文法で系統的に美しいグラフを描くことを目的として開発されているパッケージである。基本設計は確定しているが、細かい部分は現在も頻繁に開発が進められている。用意されている関数の細かな情報については、

<http://docs.ggplot2.org/>

に詳しい例題とともにまとめられている。また、良く使われる関数については、簡潔に纏めた 2 頁のシート

<http://www.rstudio.com/wp-content/uploads/2015/12/ggplot2-cheatsheet-2.0.pdf>

が用意されているので、興味に応じて参照してほしい。

4.9.3 基本的な文法

`ggplot2` では、どのデータを対象とし、どの変数を座標とし、どのような図を描くのかを順に指定するといった思想で、文法が設計されている。

まず、どのデータを対象とするかを指定すると同時に、2 次元のグラフの x 軸(横軸)と y 軸(縦軸)に何を用いるかを指定する必要がある。いくつか方法は用意されているが、関数 `ggplot2::ggplot()` を用いるのが標準である。関数 `ggplot2::ggplot()` を用いる場合には、データフレームを渡すとともに、関数 `aes()` を用いて x 座標と y 座標に対応する変数を指定することができる。関数 `ggplot2::ggplot()` だけでは何も描画はされないが、以降の描画ではこれらが既定値(指定しなくとも自動的に用いられる値)として使われることになる。この後、どのようなグラフを描くかは描画の内容を指示する関数を付与(+で加えていく)して指定するのが基本的な文法となる。

4.9.4 散布図

最も基本的な図は 2 次元の散布図(scatterplot)であるが、これには指定したデータ点を描画する関数 `ggplot2::geom_point()` を用いる。点の大きさや色も座標の一種と考えることができ、これらを使うと 2 次元以上の情報を視覚化することもできる。また、図のタイトルなどは関数 `labs()` を用いて指定することができる。

4 データのプロット

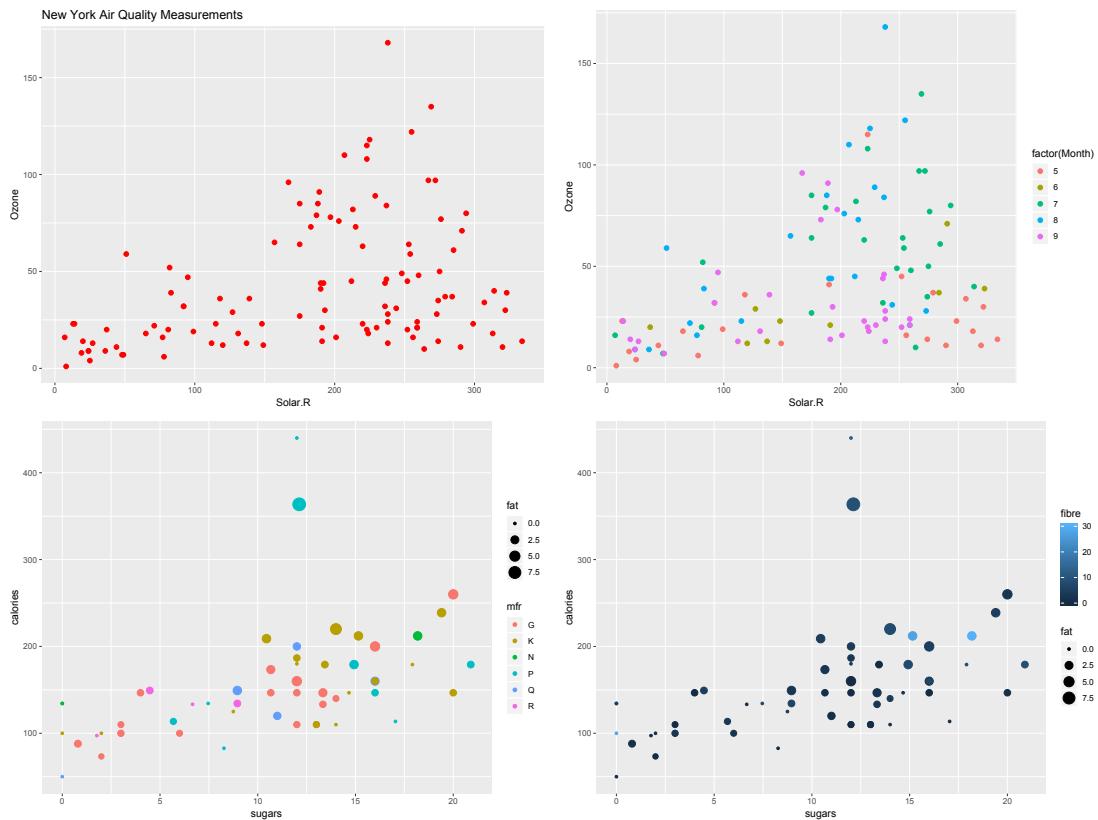


Figure 4.10: 関数 `ggplot2::geom_point()` の描画例

[Figure 4.10 を参照]

```
> ### datasets::airquality を用いた描画例
> require(MASS) # パッケージの読み込み
> require(tidyverse) # 読み込む順番にも注意が必要
> ### 日射量とオゾン量の関係を見ために散布図を描く
> ggplot(airquality, aes(Solar.R, Ozone)) +
+   geom_point(colour="red", size=2, na.rm=TRUE) + # 点の色とサイズを指定
+   labs(title="New York Air Quality Measurements") # タイトルを追加
> ### 月毎に色分けする
> ggplot(airquality, aes(Solar.R, Ozone)) +
+   geom_point(aes(colour=factor(Month)), size=2, na.rm=TRUE)
> ### MASS::UScereal を用いた描画例
> ## 会社別に糖分、カロリー、脂肪分の関係性を見る
> ggplot(UScereal, aes(sugars, calories)) +
+   geom_point(aes(size=fat, colour=mfr)) # 脂肪分の違いを点のサイズで表示
> ## 糖分、カロリー、脂肪分、繊維質の関係性を見る
> ggplot(UScereal, aes(sugars, calories)) +
+   geom_point(aes(size=fat, colour=fibre)) # 繊維質の量を点の色で表示
```

(ggplot-point.r)

4.9.5 曲線あてはめ

データの x 座標と y 座標との間にある関係を視覚化するには、曲線あてはめを行うのが簡便である。データ点に適当な方法であてはめた曲線は関数 `ggplot2::geom_smooth()` によつ

4 データのプロット

て描くことができる。既定値では loess 法を用いた曲線と標準誤差から求められる信頼区間が描かれる。オプション `method` に "lm" を指定することによって直線あてはめ(線形回帰)を行うこともできる。

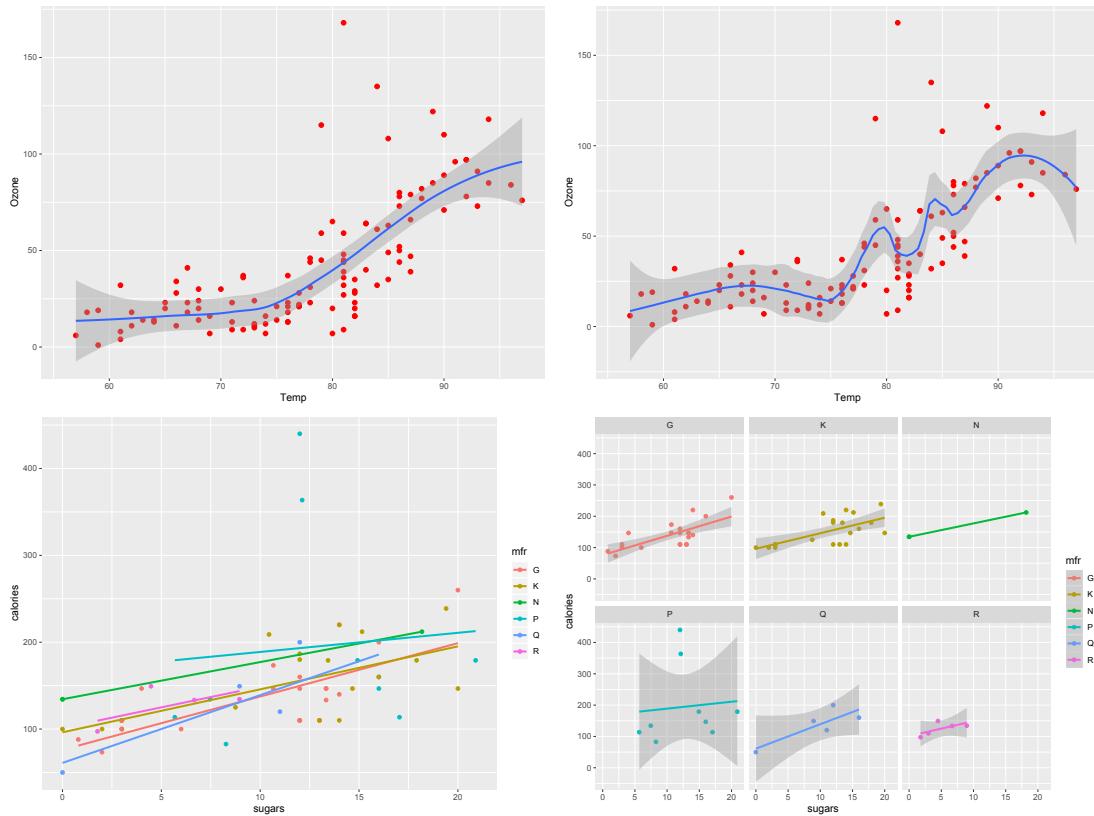


Figure 4.11: 関数 `ggplot2::geom_smooth()` の描画例

[Figure 4.11 を参照]

```
> #### datasets::airquality を用いた描画例
> ## 温度とオゾン量の関係を回帰曲線を描く
> ggplot(airquality, aes(Temp, Ozone)) +
+   geom_point(colour="red", size=2, na.rm=TRUE) +
+   geom_smooth(na.rm=TRUE)
> ## 曲線の滑らかさを変える
> ggplot(airquality, aes(Temp, Ozone)) +
+   geom_point(colour="red", size=2, na.rm=TRUE) +
+   geom_smooth(span=0.3, na.rm=TRUE) # 幅の狭い窓で平均化
> ### MASS::UScereal を用いた描画例
> ## 会社毎の糖分とカロリーの回帰直線を見る
> ggplot(UScereal, aes(sugars, calories, colour=mfr)) +
+   geom_point() +
+   geom_smooth(method="lm", se=FALSE) # 信頼区間を表示しない
> ## 信頼区間を付けて別々に表示する
> ggplot(UScereal, aes(sugars, calories, colour=mfr)) +
+   geom_point() +
+   geom_smooth(method="lm") +
+   facet_wrap(~mfr) # 会社ごとに別のグラフを作成
```

(ggplot-smooth.r)

4 データのプロット

4.9.6 ヒストグラム

1次元データの分布の概形を捉えるにはヒストグラム (histogram) が重要であるが、関数 `ggplot2::geom_histogram()` を用いることによって描画することができる。また、棒状のグラフではなく折れ線で描くこともでき、これには関数 `ggplot2::geom_freqpoly()` を用いる。なお、 y 軸の値は自動的に計算されるので特に指定する必要はないが、複数の値を比較する場合には密度に正規化して表示した方が良いこともある。内部で計算された密度の値 (density) を y 軸に指定するには `..density..` を指定する。

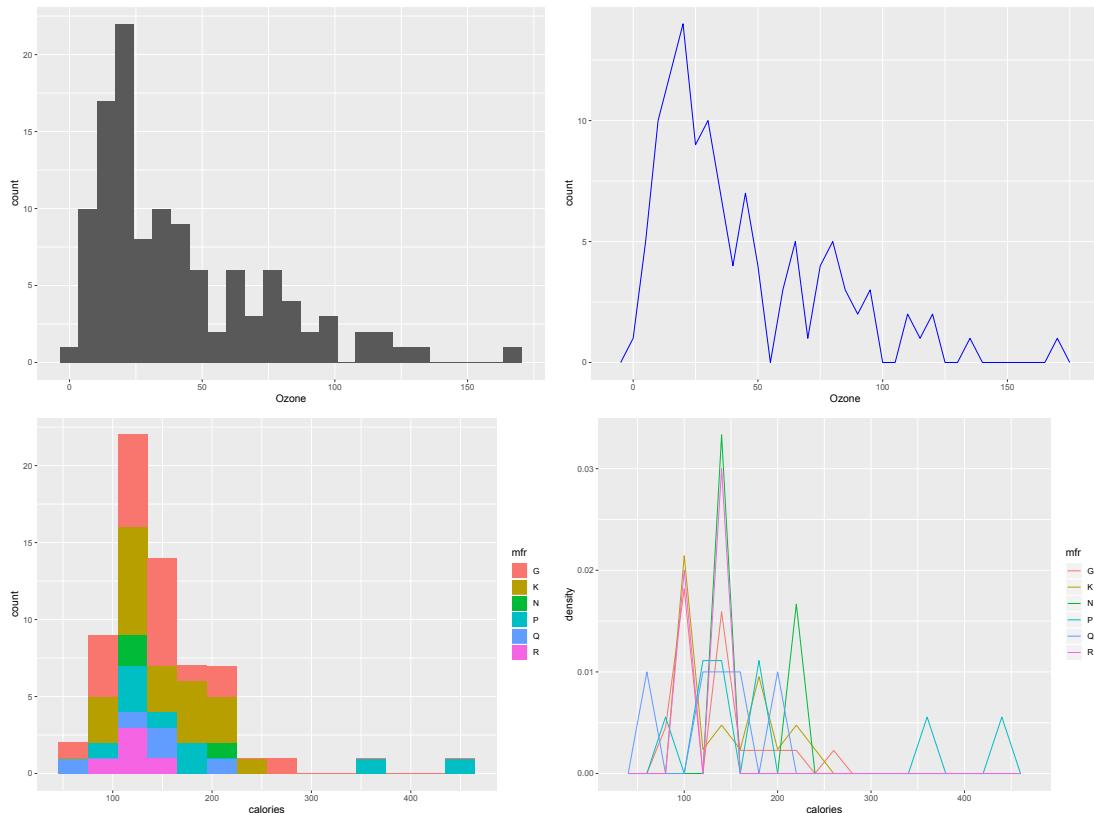


Figure 4.12: 関数 `ggplot2::geom_histogram()` の描画例

[Figure 4.12 を参照]

```
> ### datasets::airquality を用いた描画例
> ## オゾン量のヒストグラムを描く
> ggplot(airquality, aes(Ozone)) +
+   geom_histogram(bins=25, na.rm=TRUE) # ビンの数を指定
> ## オゾン量のヒストグラムを折れ線で描く
> ggplot(airquality, aes(Ozone)) +
+   geom_freqpoly(binwidth=5, colour="blue", na.rm=TRUE) # ビンの幅を指定
> ### MASS::UScereal を用いた描画例
> ## 会社毎のカロリーを詰み上げてヒストグラムを描く
> ggplot(UScereal, aes(calories)) +
+   geom_histogram(binwidth=30, aes(fill=mfr))
> ## 会社別の頻度(..density..)を折れ線で描く
> ggplot(UScereal, aes(calories, ..density.., colour=mfr)) +
+   geom_freqpoly(binwidth=20)
```

(ggplot-hist.r)

4.9.7 箱ひげ図

ヒストグラムより簡便に分布の様子を視覚化する方法として箱ひげ図 (boxplot) がある。これは関数 `ggplot2::geom_boxplot()` を用いて描くことができる。

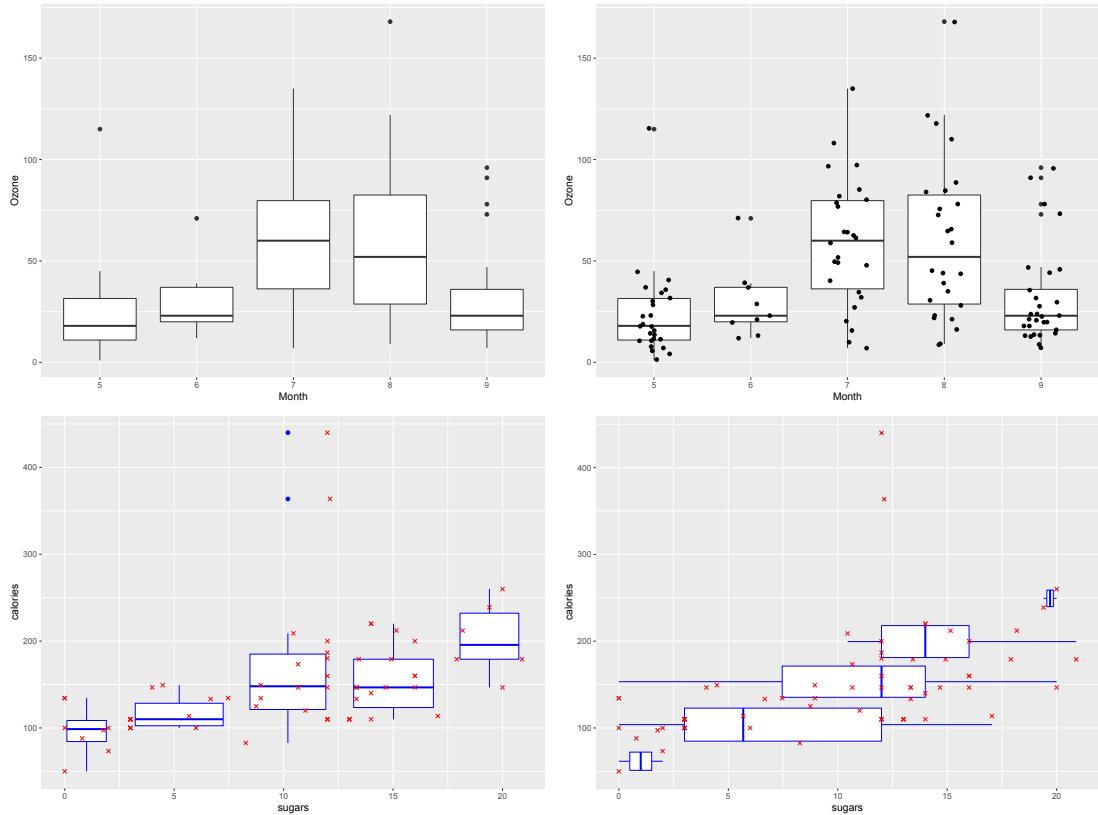


Figure 4.13: 関数 `ggplot2::geom_boxplot()` の描画例

[Figure 4.13 を参照]

```
> ### datasets::airquality を用いた描画例
> ## 月別のオゾン量の箱ひげ図を描く
> ggplot(airquality, aes(factor(Month), Ozone)) +
+   geom_boxplot(na.rm=TRUE) +
+   labs("x"="Month") # x 軸のラベルを変更
> ## データを付随させる
> ggplot(airquality, aes(factor(Month), Ozone)) +
+   geom_boxplot(na.rm=TRUE) +
+   geom_jitter(width=0.2, na.rm=TRUE) + # データを重ならないように表示
+   labs("x"="Month")
> ### MASS::UScereal を用いた描画例
> ## 糖分とカロリーの関係性を見る
> ggplot(UScereal, aes(sugars, calories)) +
+   geom_boxplot(aes(group=cut_width(sugars,5)), colour="blue") +
+   geom_point(colour="red", shape=4)
> ## 軸を入れ換えて箱ひげ図を描く
> ggplot(UScereal, aes(calories, sugars)) + # xyを入れ換えておく
+   geom_boxplot(aes(group=cut_width(calories,50)), colour="blue") +
+   geom_point(colour="red", shape=4) +
+   coord_flip() # 軸の入れ換え
```

(ggplot-boxplot.r)

4.9.8 折れ線グラフ

折れ線グラフは時系列など x 軸の増加に伴なう y 軸の変動を視覚化する場合に用いられる。点列を結ぶ線を描くには関数 `ggplot2::geom_line()` を用いる。なお、時系列を図示する際に、データフレームに時間の情報が不足している場合には自分で時間に関する変数を整理し直さなくてはならないので、注意が必要である。

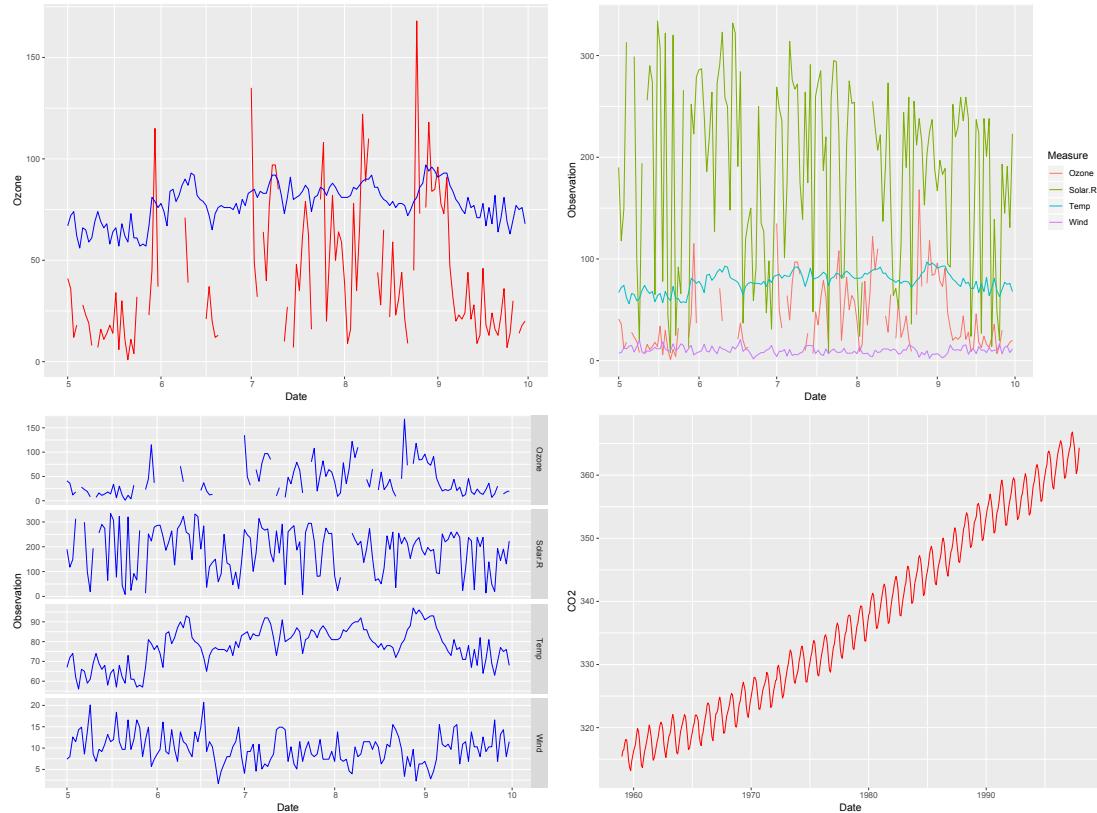


Figure 4.14: 関数 `ggplot2::geom_line()` の描画例

[Figure 4.14 を参照]

```
> #### datasets::airquality を用いた描画例
> ## オゾン量を時系列としてグラフを描く
> mydat <- airquality %>%
+     ## 時間の情報を整理して列を作成
+     mutate(Date=as.Date(paste(Month, Day, "73", sep="/"), "%m/%d/%y")) %>%
+     select(Date, Ozone:Temp)
> ggplot(mydat, aes(x=Date)) +
+     geom_line(aes(y=Ozone), colour="red") +
+     geom_line(aes(y=Temp), colour="blue") # 表示したいデータを選択
> ## 複数の系列を描画
> require(tidyverse)
> mydat <- airquality %>%
+     mutate(Date=as.Date(paste(Month, Day, "73", sep="/"), "%m/%d/%y")) %>%
```

4 データのプロット

```

+   select(Ozone:Temp,Date) %>%
+   gather(Measure, Observation, Ozone:Temp)
> ## ggplot(mydat, aes(Date, Observation)) +
+   geom_line(aes(colour=Measure)) # 計測データ毎に色を変えて表示
> ## 別の facet に表示
> ggplot(mydat, aes(Date, Observation)) +
+   geom_line(colour="blue") +
+   ## 計測データ毎に別のグラフ (facet) で表示
+   facet_grid(Measure ~ ., scales="free_y") # y座標を適宜スケーリング
> #### datasets::co2 を用いた描画例
> ## ts class のデータを変換して描画
> mydat <- data.frame(CO2=as.vector(co2), Date=as.vector(time(co2)))
> ggplot(mydat, aes(Date, CO2)) +
+   geom_line(colour="red")

```

(ggplot-line.r)

4.9.9 棒グラフ

棒グラフは集計されたデータの比較を視覚的に行う場合に重要であるが、これは関数 `ggplot2::geom_bar()` を用いて描画することができる。データフレームの集計は、前章で用いたパッケージ `dplyr`などを用いて行うことになる。

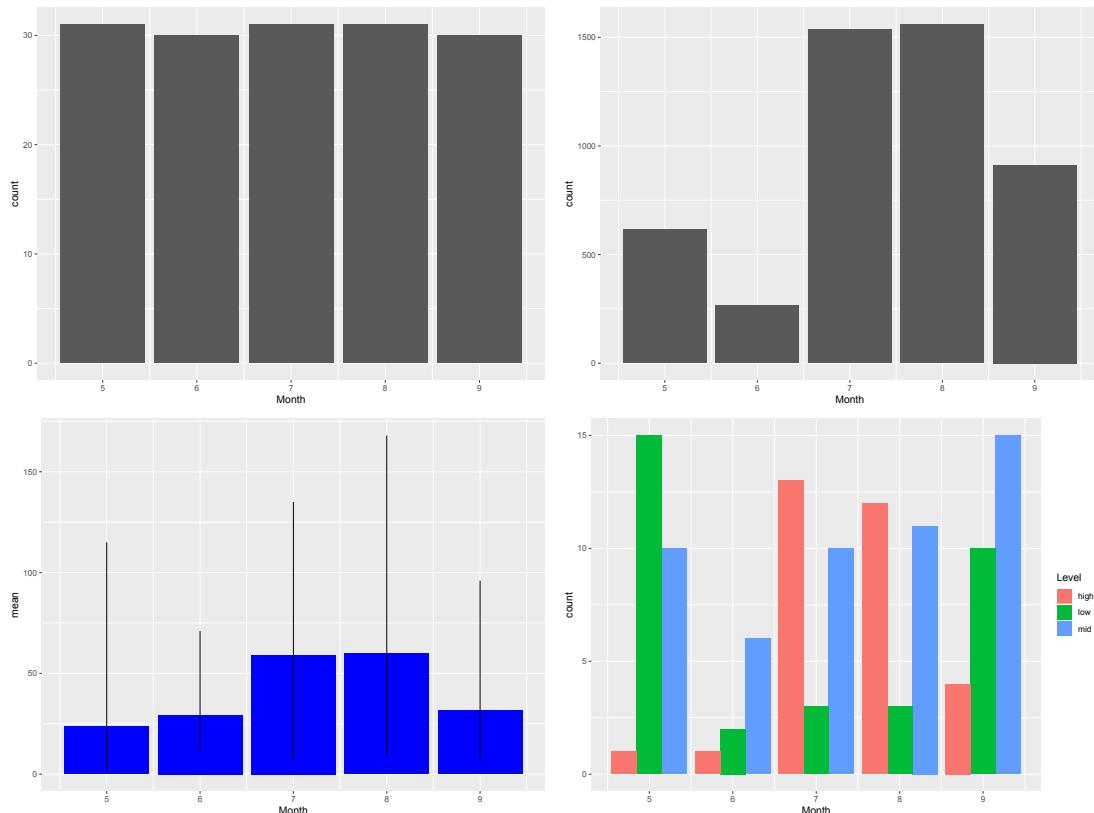


Figure 4.15: 関数 `ggplot2::geom_bar()` の描画例

4 データのプロット

[Figure 4.15 を参照]

```
> ### datasets::airquality を用いた描画例
> ## 月別の計測日数を棒グラフとして表示
> ggplot(airquality, aes(Month)) +
+   geom_bar()
> ## 月別のオゾン量の合計を表示
> ggplot(airquality, aes(Month)) +
+   geom_bar(aes(weight=Ozone))
> ## 月別にオゾン量を集計
> mydat <- airquality %>%
+   group_by(Month) %>%
+   summarize(
+     mean      = mean(Ozone, na.rm=TRUE),
+     min       = min(Ozone, na.rm=TRUE),
+     max       = max(Ozone, na.rm=TRUE))
> ## 月別のオゾン量の平均を表示
> ggplot(mydat, aes(Month, mean)) +
+   geom_bar(stat="identity", fill="blue") +
+   geom_linerange(aes(ymin=min,ymax=max)) # 値域も表示
> ## オゾン量の高低のラベルを付加
> mydat <- airquality %>%
+   mutate(Level=ifelse(Ozone>60,"high",ifelse(Ozone<20,"low","mid"))) %>%
+   filter(!is.na(Level)) # Level が計算できなかったものを除く
> ## 月別のオゾン量の高低の割合を表示
> ggplot(mydat, aes(Month)) +
+   geom_bar(aes(fill = Level), position = "dodge")
```

(ggplot-bar.r)

5 モンテカルロ法

現実の世界には確率的な取り扱いが必要な事象があり、それらを確率的現象という。現実のデータに含まれる不確定性は確率的現象によって生まれる。確率的現象にはさまざまなものがあり、中には我々の直感と大きく異なる現象も含まれる。確率的現象は、問題を抽象化・単純化して理論的な解析を詳しく行うことが可能な場合もあるが、理論的に解析を行うことが難しい現象も数多く存在する。そうした複雑な問題に対して、計算機上の擬似乱数を利用して数値的に現象を再現し、その性質を調べる方法がある。それは**モンテカルロ法**(Monte-Carlo method)あるいは**確率シミュレーション**(stochastic simulation)と呼ばれる。計算機上では繰り返しシミュレーションを行うことができるので、原因となる要素が変化すると結果にどのような影響を及ぼすかを詳細に調べることができる。この章では簡単な例を取り上げながらモンテカルロ法の考え方について紹介する。

5.1 亂数

亂数とはランダムに生成された数列のことである。もちろんコンピューターでは完全にランダムに数字を発生させることは不可能なため、それらの乱数は厳密には**擬似乱数**と呼ばれる¹。特に数値シミュレーションを行う上では、それが再現可能であることが要請されるため、発生される乱数も再現可能である必要がある。Rではこれを実行するために、乱数の初期値を指定するための関数 `set.seed()` が用意されている（同一の初期値から生成される乱数は同一のものとなる）。

ここでは基本的な乱数として、ランダムサンプリング、二項乱数および一様乱数を考える。ランダムサンプリングは、その名の通り「与えられた集合の要素をランダムに抽出することで発生」する乱数のことである。二項乱数は、「確率 p で表がでるコインを n 回投げた際の表が出る回数」に対応する乱数である。従って p と n によって乱数の発生の仕方が変わるために、それを明示する場合は「確率 p に対する次数 n の二項乱数」と言う。一様乱数は、「ある決まった区間 (a, b) ($a < b$) に含まれる数字からランダムに発生」する乱数のことである²。従って区間 (a, b) によって乱数の発生の仕方が変わるため、それを明示する場合は「区間 (a, b) 上の一様乱数」と言う。

ランダムサンプリングは関数 `sample()` で実行できる。二項乱数および一様乱数はそれぞれ関数 `rbinom()` および `runif()` を用いて発生させる。

```
> #### 関数 sample の使い方
> x <- 1:10 # サンプリング対象の集合をベクトルとして定義
> set.seed(123) # 亂数のシード値(任意に決めてよい)を指定
> sample(x, 5) # x から 5 つの要素を重複なしでランダムに抽出
[1] 3 8 4 7 6
> sample(x, length(x)) # x の要素のランダムな並べ替えとなる
[1] 1 5 8 4 3 9 2 6 7 10
```

¹R では擬似乱数を発生させるための方法として“Mersenne-Twister”がデフォルトでは用いられている。
`help(Random)` 参照。

² (a, b) は a より大きく b より小さい実数全体からなる集合を表す。

```

> sample(x, 5, replace=TRUE) # x から 5つの要素を重複ありでランダムに抽出
[1] 9 3 1 4 10
> sample(1:6, 10, replace=TRUE) # サイコロを 10回振る実験の再現
[1] 6 5 4 6 4 5 4 4 2 1
> sample(1:6, 10, prob=6:1, replace=TRUE) # 出る目の確率に偏りがある場合
[1] 6 5 3 4 1 2 4 1 2 1
> ### 関数 rbinom の使い方
> rbinom(10, size=4, prob=0.5) # 確率 0.5 に対する次数 4 の二項乱数を 10 個発生
[1] 1 2 2 2 1 1 1 2 1 3
> rbinom(20, size=4, prob=0.2) # 個数を 20, 確率を 0.2 に変更
[1] 0 1 1 0 1 0 0 1 2 0 1 0 0 0 1 1 1 1 1
> ### 関数 runif の使い方
> runif(5, min=-1, max=2) # 区間 (-1,2) 上の一様乱数を 5 個発生
[1] 1.2634255 0.8876634 1.1305472 -0.9981257 0.4259497
> runif(5) # 指定しない場合は区間 (0,1) が既定値
[1] 0.2201189 0.3798165 0.6127710 0.3517979 0.1111354
> ### 関数 set.seed について
> set.seed(1) # 亂数の初期値を seed=1 で指定
> runif(5)
[1] 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819
> set.seed(2) # 亂数の初期値を seed=2 で指定
> runif(5) # seed=1 の場合と異なる結果
[1] 0.1848823 0.7023740 0.5733263 0.1680519 0.9438393
> set.seed(1) # 亂数の初期値を seed=1 で指定
> runif(5) # 初めの seed=1 の場合と同じ結果
[1] 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819

```

(mc-sample.r)

R には他にも様々な種類の確率分布に従う乱数が実装されている。

5.2 数値シミュレーション

以下では具体的な例題を用いて確率的なシミュレーションを説明する。

5.2.1 コイン投げの賭け

まず初めに次の簡単な問題を考えてみよう。

演習 5.1. A と B の二人で交互にコインを投げる。最初に表が出た方を勝ちとするとき、A と B それぞれの勝率はいくつとなるか？

5 モンテカルロ法

コインを投げる試行は Bernoulli 分布(サイズ 1 の 2 項分布)なので、乱数生成には関数 `rbinom()` を利用することができる。

別の方法としては関数 `runif()` を利用して生成した乱数が $1/2$ 以上であるかどうかで模擬することもできる。

シミュレーションの一例を以下に示す。

```
> #### コイン投げの賭け
>
> ## コイン投げの試行
> ## いろいろな書き方があるが、まずはベタに書いてみる
> myTrial <- function(){ # 名前は何でも良い
+   while(TRUE){ # 永久に回るループ
+     if(rbinom(1,size=1,prob=0.5)==1){
+       return("A") # A が表を出して終了
+     }
+     if(rbinom(1,size=1,prob=0.5)==1){
+       return("B") # B が表を出して終了
+     }
+     ## どちらも裏ならもう一度ループ
+   }
+ }
> ## 試行を行ってみる
> myTrial()

[1] "A"

> myTrial()

[1] "A"

> myTrial()

[1] "A"

> ## Monte-Carlo simulation
> ## set.seed(8888) # 実験を再現したい場合はシードを指定する
> mc <- 10000 # 回数を設定
> myData <- replicate(mc,myTrial())
> ## 簡単な集計
> table(myData) # 頻度

myData
  A     B
6705 3295

> table(myData)/mc # 確率(推定値)

myData
  A     B
0.6705 0.3295
```

(mc-coin.r)

5.2.2 Buffon の針

次に 18 世紀の学者 Buffon による有名な問題を考えてみよう。

演習 5.2. 2 次元平面上に等間隔 d で平行線が引いてある。長さ l の針($l < d$ とする)をこの平面上にランダム(でたらめ)に落としたとき、平行線と交わる確率はいくつか？

5 モンテカルロ法

針の中心位置と一番近い平行線を原点とし、水平方向の座標を x とする。問題の繰り返し構造に注意すれば、 x は区間 $[-d/2, d/2]$ で一様に分布する（どの点が得られるかは無作為）と考えられる。また針に向きがあるとして、針が図の水平方向となす角度を θ とする。 θ も同様に区間 $[0, 2\pi]$ で一様に分布すると考えられる。この試行の見本点は (x, θ) で表され、その見本空間は

$$\Omega = [-d/2, d/2] \times [0, 2\pi] \subset \mathbb{R}^2$$

となる。また、針が平行線と交わる条件は、針の両端の座標の符号が異なること

$$\left(x + \frac{l}{2} \cos \theta \right) \left(x - \frac{l}{2} \cos \theta \right) < 0$$

で表される。

上記の条件を満たす (x, θ) の領域を考えると

$$P(\text{針が平行線と交わる}) = \frac{4l}{2\pi d} = \frac{2l}{\pi d}$$

となる。

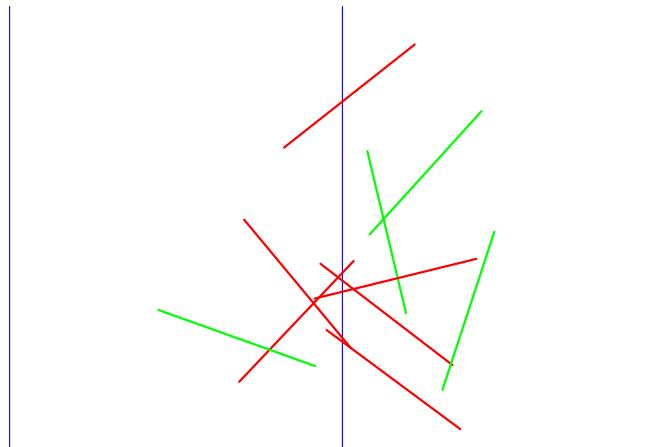


Figure 5.1: 針と平行線の関係を図示した例

[Figure 5.1 を参照]

```
> ### Buffon の針
>
> ## 針を投げる試行
> myTrial <- function(d, l, verbose=FALSE){ # d と l を指定する
+   x <- runif(1, min=-d/2, max=d/2)
+   theta <- runif(1, min=0, max=2*pi)
+   cross <- FALSE # 交わったかどうかを示す変数
+   if((x+l*cos(theta)/2)*(x-l*cos(theta)/2)<0){
+     cross <- TRUE # 交わった場合に書き換え
+   }
}
```

5 モンテカルロ法

```

+     if(verbose==TRUE){
+         return(c(x=x,theta=theta,cross=as.numeric(cross)))
+     } else {
+         return(cross)
+     }
+ }
> ## 試行を行ってみる
> d <- 10
> l <- 5
> myTrial(d,l,verbose=TRUE)

      x      theta      cross
-2.592678  1.323649  0.000000

> myTrial(d,l,verbose=TRUE)

      x      theta      cross
3.868208  2.642052  0.000000

> ## 絵にしてみる
> plot(c(0,0),type="n", # 空のキャンバスを作る
+       xlim=c(-d,d),asp=1,ann=FALSE,axes=FALSE)
> abline(v=c(-10,0,10),col="blue") # 線を引く
> for (i in 1:10) {
+   obs <- myTrial(d,l,verbose=TRUE)
+   x <- obs["x"]
+   theta <- obs["theta"]
+   y <- runif(1,min=-d/2,max=d/2) # y 座標は適当にばらまく
+   x1 <- x-l/2*cos(theta)
+   x2 <- x+l/2*cos(theta)
+   y1 <- y-l/2*sin(theta)
+   y2 <- y+l/2*sin(theta)
+   lines(c(x1,x2),c(y1,y2),
+         col=ifelse(x1*x2<0,"red","green"),
+         lty = "solid", lwd = 2)
+ }
> ## Monte-Carlo simulation
> ## set.seed(8888) # 実験を再現したい場合はシードを指定する
> mc <- 10000 # 回数を設定
> myData <- replicate(mc,myTrial(d,l))
> ## 簡単な集計
> table(myData) # 頻度 (TRUE が針の交わった回数)

myData
FALSE  TRUE
 6832  3168

> table(myData)/mc # 確率 (推定値)

myData
FALSE  TRUE
 0.6832  0.3168

> print((2*l)/(pi*d)) # 針の交わる確率 (理論値)

[1] 0.3183099

```

(mc-buffon.r)

5.2.3 Monty Hall 問題

次の問題はアメリカの雑誌で、その解をめぐって大議論に発展した問題である。

演習 5.3. プレーヤーの前に閉まった3つのドアがある。1つのドアの後ろには景品の新車が、2つのドアの後ろにははずれを意味するヤギがいる。プレーヤーは新車のドアを当てることができると景品として新車がもらえる。プレーヤーが1つのドアを選択した後、司会のモンティが残りのドアのうちヤギがいるドアを開けてヤギを見せる。(プレーヤーがどのドアを選んでも、モンティはヤギのいるドアを開けられることに注意せよ。) ここでプレーヤーは、最初に選んだドアを、開けられていない残ったドアに変更してもよいと言われる。プレーヤーはドアを変更すべきだろうか？

実際にシミュレーションを行うと以下のようになり、最初の選択と変更した場合で景品を貰える確率が異なることがわかる。

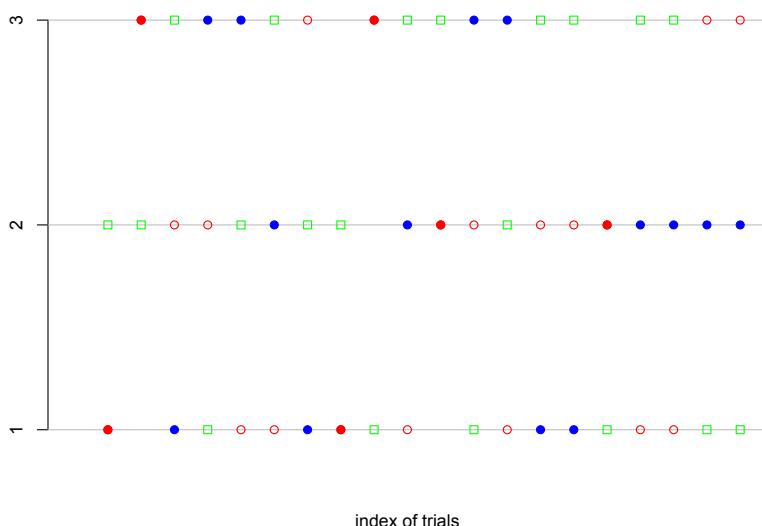


Figure 5.2: シミュレーション例: 赤○は最初に選択したドアの位置、緑四角は開けられたドアの位置、赤塗り潰しは最初の選択のままが正解の場合、青塗り潰しはドアを変えるのが正解の場合。

[Figure 5.2 を参照]

```
> ### Monty Hole 問題
>
> ## クイズに答える試行
> myTrial <- function(verbose=FALSE){
+   prize <- sample(1:3,size=1) # 賞品の置かれた扉
+   choice <- sample(1:3,size=1) # 最初の選択
+   if(prize==choice) {
+     win <- "stay"
+     door <- sample(setdiff(1:3,prize),size=1)
+   } else {
+     win <- "change"
+     door <- setdiff(1:3,union(prize,choice))
+   }
}
```

```

+   if(verbose==TRUE){
+     return(c(prize=prize,choice=choice,door=door))
+   } else {
+     return(win)
+   }
+ }
> ## 試行を行ってみる
> myTrial()

[1] "change"

> myTrial(verbose=TRUE)

prize choice door
      2       1      3

> ## 絵にしてみる
> mc <- 20
> plot(c(0,0),type="n", # 空のキャンバスを作る
+       xlim=c(0,mc),ylim=c(1,3),ann=FALSE,axes=FALSE)
> title(xlab="index of trials")
> axis(2,at=1:3,labels=1:3)
> abline(h=1:3,col="grey") # 線を引く
> for (i in 1:mc) {
+   obs <- myTrial(verbose=TRUE)
+   prize <- obs["prize"]
+   choice <- obs["choice"]
+   door <- obs["door"]
+   points(i,door,pch=0,col="green")
+   points(i,prize,pch=1,col="red")
+   points(i,choice,pch=19,col=ifelse(prize==choice,"red","blue"))
+ }
> ## Monte-Carlo simulation
> ## set.seed(8888) # 実験を再現したい場合はシードを指定する
> mc <- 10000 # 回数を設定
> myData <- replicate(mc,myTrial())
> ## 簡単な集計
> table(myData) # 頻度

myData
change stay
 6671 3329

> table(myData)/mc # 確率 (推定値)

myData
change stay
0.6671 0.3329

```

(mc-montyhole.r)

5.2.4 St Petersburg のパラドックス

次の例は、無限回の試行を行う理論と、有限回しか実行できない現実との関係を考える問題である。

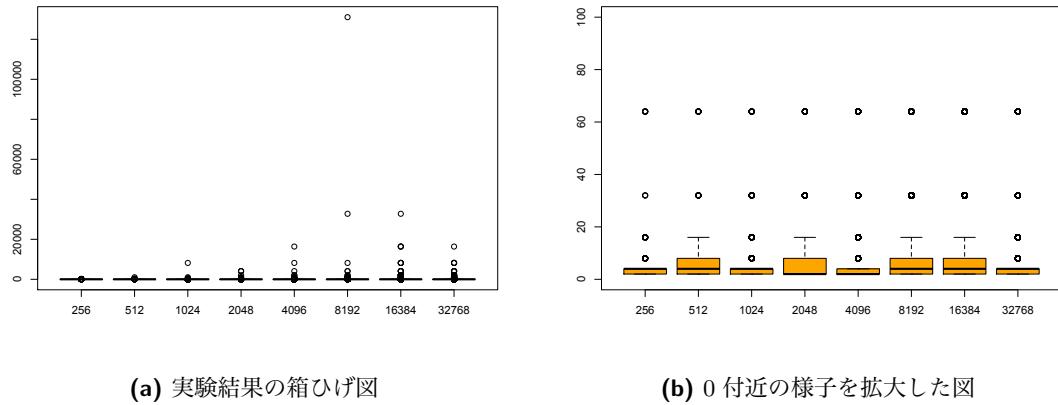
演習 5.4. 偏りのないコインを表が出るまで投げ続け、賞金を貰うゲームを考える。表が出るまでにコインを投げた回数が n 回であるとき、貰える賞金は 2^n 円とする。このとき賞金の期待値は

$$\mathbb{E}[\text{賞金}] = 2 \times \frac{1}{2} + 2^2 \times \frac{1}{2^2} + 2^3 \times \frac{1}{2^3} + \dots = \infty$$

5 モンテカルロ法

となるが、ゲームを行う回数は現実には有限回であるが、それでも期待値は発散すると考えて良いであろうか？

ゲームを繰り返し無限回行う場合には上記の期待値の計算は正しいが、実際に無限回行うこととはできない（例えば人には寿命がある）。有限回（例えば100回）でやめざるを得ないときに、実際の「期待値」はいくつになるか求めてみよう。



(a) 実験結果の箱ひげ図

(b) 0付近の様子を拡大した図

Figure 5.3: シミュレーションの例

[Figure 5.3 を参照]

```
> #### St Petersburg のパラドックス
>
> ## コイン投げの賭けの試行
> myTrial <- function(verbose=FALSE){
+   num <- 1 # コイン投げの回数 (次に投げるコインが何回目か)
+   while(TRUE){ # 条件が満たされるまでループする
+     if(runif(1)>0.5) break # 一様分布で模擬
+     ## if(rbinom(1,size=1,prob=0.5)==1) break
+     num <- num + 1 # 回数を増やす
+   }
+   bounty <- 2^num
+   if(verbose==TRUE){
+     return(c(num=num,bounty=bounty))
+   } else {
+     return(bounty)
+   }
+ }
> ## 試行を行ってみる
> myTrial()
[1] 2
> myTrial(verbose=TRUE)
  num  bounty
  3       8

> ## Monte-Carlo simulation
> ## set.seed(8888) # 実験を再現したい場合はシードを指定する
> myDF <- data.frame(mc=NULL,bounty=NULL)
> for (mc in c(2^(8:15))) {
```

```

+   myData <- replicate(mc,myTrial())
+   cat("試行回数: ", mc, "\n")
+   cat("平均値: ", mean(myData), "\n")
+   myDF <- rbind(myDF,data.frame(mc=rep(mc,mc),bounty=myData))
+
試行回数: 256
平均値: 7.828125
試行回数: 512
平均値: 12.1875
試行回数: 1024
平均値: 18.10742
試行回数: 2048
平均値: 16.87988
試行回数: 4096
平均値: 17.27588
試行回数: 8192
平均値: 32.61719
試行回数: 16384
平均値: 19.42212
試行回数: 32768
平均値: 13.56067

> boxplot(bounty ~ mc, data=myDF) # 回数ごとの箱ひげ図を表示
> boxplot(bounty ~ mc, data=myDF, ylim=c(0,100), col="orange")

```

(mc-stpetersburg.r)

5.2.5 秘書問題

最後は、現実にありえる問題を単純化して数学的にも扱い易くしたものである。

演習 5.5. 秘書を 1 人雇いたいとする。前提条件は以下のとおりである。

1. n 人が応募しており、 n は既知とする。
2. 応募者には 1 位から n 位まで同順位無しで順位付けできる。
3. 無作為な順序で 1 人ずつ面接を行う。
4. 毎回の面接後、その応募者を採用するか否かを決定する。
5. 不採用にした応募者を後から採用することはできない。

これに対して「面接者は最初の $r - 1$ 人の応募者をスキップし、その次の応募者がそれまで面接した中で最もよい応募者なら採用する」という戦略を取るとき、最良の応募者を採用する確率を最も高くするためには r をいくつとすれば良いか？

これは最適停止問題とよばれる最適戦略を問う問題の一種である。問題の条件によっていろいろな戦略が考えられているが、上記は最も単純なものである。

[Figure 5.4 を参照]

```

> ### 秘書問題
>
> ## 秘書の採用の試行
> myTrial <- function(n,r,verbose=FALSE){ # n と r を指定

```

5 モンテカルロ法

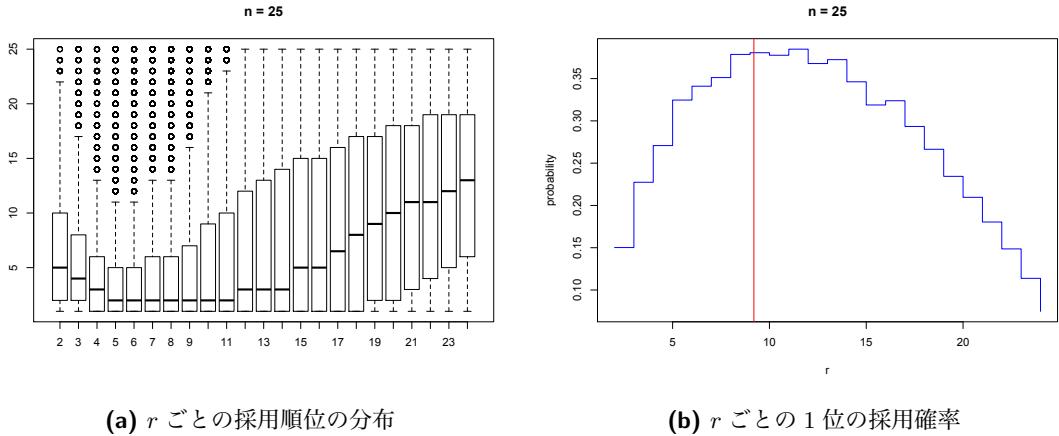


Figure 5.4: シミュレーションの例

```

+ applicants <- sample(1:n,size=n)
+ ref <- applicants[1:(r-1)]
+ test <- applicants[r:n]
+ idx <- which(test < min(ref))
+ if(length(idx)==0) {
+   employed <- applicants[n]
+ } else {
+   employed <- test[idx[1]]
+ }
+ if(verbose==TRUE){
+   return(list(applicants=applicants,employed=employed))
+ } else {
+   return(employed)
+ }
+ }
> ## 試行を行ってみる
> n <- 10
> myTrial(n,2,verbose=TRUE)

$applicants
[1] 2 7 9 1 8 4 3 6 5 10

$employed
[1] 1

> myTrial(n,3,verbose=TRUE)

$applicants
[1] 7 3 9 10 5 8 4 2 6 1

$employed
[1] 2

> myTrial(n,4,verbose=TRUE)

$applicants
[1] 2 9 10 7 8 1 6 5 4 3

$employed
[1] 1

> myTrial(n,5,verbose=TRUE)

```

5 モンテカルロ法

```
$applicants
[1] 3 9 6 10 2 5 7 1 8 4

$employed
[1] 2

> myTrial(n, 6, verbose=TRUE)

$applicants
[1] 9 4 5 10 1 3 7 2 6 8

$employed
[1] 8

> ## Monte-Carlo simulation
> ## set.seed(8888) # 実験を再現したい場合はシードを指定する
> mc <- 5000
> n <- 25 # 候補者数を変えて実験
> myDF <- data.frame(r=NULL, employed=NULL)
> for (r in 2:(n-1)) {
+   myData <- replicate(mc, myTrial(n, r))
+   cat("試行回数: ", r, "\n")
+   print(table(myData))
+   myDF <- rbind(myDF, data.frame(r=rep(r, mc), employed=myData))
+ }

試行回数: 2
myData
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
751 601 463 383 339 315 275 262 220 206 170 162 160 121 101 106 88 64 62 45
 21 22 23 24 25
 35 22 29 15  5
試行回数: 3
myData
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1137 741 562 425 371 282 231 186 190 140 132 116 75 77 73 51
 17 18 19 20 21 22 23 24 25
 39 31 28 27 20 22 12 15 17
試行回数: 4
myData
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1354 829 559 417 350 246 199 164 138 126 80 72 72 52 45 37
 17 18 19 20 21 22 23 24 25
 40 42 20 22 32 34 28 23 19
試行回数: 5
myData
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1623 879 564 414 280 199 150 118 92 85 62 48 32 46 44 35
 17 18 19 20 21 22 23 24 25
 39 35 38 34 36 40 36 28 43
試行回数: 6
myData
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1705 959 538 353 266 172 140 102 84 60 59 41 43 43 37 35
 17 18 19 20 21 22 23 24 25
 42 25 42 50 50 40 32 44 38
試行回数: 7
myData
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1756 915 508 341 219 154 98 95 72 76 57 46 53 43 51 64
 17 18 19 20 21 22 23 24 25
```

5 モンテカルロ法

45	54	41	43	56	50	51	49	63
試行回数: 8								
myData								
1 2 3 4 5 6 7 8 9								
1893	822	486	299	175	134	94	93	73
17	18	19	20	21	22	23	24	25
57	45	65	55	49	53	76	64	61
試行回数: 9								
myData								
1 2 3 4 5 6 7 8 9								
1903	843	441	261	155	114	88	76	71
17	18	19	20	21	22	23	24	25
63	68	62	62	58	70	58	70	59
試行回数: 10								
myData								
1 2 3 4 5 6 7 8 9								
1888	791	422	211	151	87	87	88	81
17	18	19	20	21	22	23	24	25
75	77	73	76	91	75	82	91	64
試行回数: 11								
myData								
1 2 3 4 5 6 7 8 9								
1924	709	341	200	136	118	90	94	88
17	18	19	20	21	22	23	24	25
64	66	88	65	78	79	98	73	98
試行回数: 12								
myData								
1 2 3 4 5 6 7 8 9								
1839	655	335	177	120	90	88	86	97
17	18	19	20	21	22	23	24	25
92	92	89	86	101	92	97	93	90
試行回数: 13								
myData								
1 2 3 4 5 6 7 8 9								
1862	612	276	160	127	101	101	102	101
17	18	19	20	21	22	23	24	25
114	99	99	92	92	116	104	84	94
試行回数: 14								
myData								
1 2 3 4 5 6 7 8 9								
1731	603	235	164	108	105	114	117	110
17	18	19	20	21	22	23	24	25
98	123	115	104	103	90	111	119	108
試行回数: 15								
myData								
1 2 3 4 5 6 7 8 9								
1594	536	220	145	142	111	139	115	125
17	18	19	20	21	22	23	24	25
124	120	108	114	137	122	115	108	128
試行回数: 16								
myData								
1 2 3 4 5 6 7 8 9								
1619	430	182	152	127	141	123	127	108
17	18	19	20	21	22	23	24	25
125	100	114	115	126	123	121	122	143
試行回数: 17								
myData								
1 2 3 4 5 6 7 8 9								
1467	386	210	156	134	147	113	128	135
17	18	19	20	21	22	23	24	25

5 モンテカルロ法

```

151 126 138 110 119 143 129 159 129
試行回数: 18
myData
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1332 353 187 129 144 156 166 132 140 145 131 152 141 148 120 139
 17 18 19 20 21 22 23 24 25
 147 125 141 123 148 152 139 161 149
試行回数: 19
myData
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1172 315 194 139 129 137 175 170 164 159 140 152 143 153 149 164
 17 18 19 20 21 22 23 24 25
 150 135 149 169 148 155 151 147 141
試行回数: 20
myData
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1048 283 178 147 166 162 155 144 147 157 152 151 182 155 153 185
 17 18 19 20 21 22 23 24 25
 154 174 154 139 172 159 155 173 155
試行回数: 21
myData
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
902 256 182 170 141 152 163 169 156 192 168 169 159 188 170 152 168 140 156 157
 21 22 23 24 25
206 149 164 164 207
試行回数: 22
myData
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
743 198 189 178 171 162 176 189 189 165 162 161 153 185 184 183 167 186 180 186
 21 22 23 24 25
161 181 176 193 182
試行回数: 23
myData
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
569 230 160 178 187 204 186 164 184 194 190 187 150 198 180 171 184 177 178 195
 21 22 23 24 25
189 192 170 182 201
試行回数: 24
myData
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
374 219 195 187 218 193 202 216 199 155 173 167 198 176 195 173 183 204 204 198
 21 22 23 24 25
190 185 204 221 171

> boxplot(employed ~ r, data=myDF, # rごとの箱ひげ図を表示
+           main=paste("n =", n))
> (myDF2 <- aggregate(myDF[, "employed"], by=list(r=myDF$r),
+                      FUN=function(x){mean(x==1)}))

      r      x
1  2 0.1502
2  3 0.2274
3  4 0.2708
4  5 0.3246
5  6 0.3410
6  7 0.3512
7  8 0.3786
8  9 0.3806
9 10 0.3776
10 11 0.3848
11 12 0.3678

```

```

12 13 0.3724
13 14 0.3462
14 15 0.3188
15 16 0.3238
16 17 0.2934
17 18 0.2664
18 19 0.2344
19 20 0.2096
20 21 0.1804
21 22 0.1486
22 23 0.1138
23 24 0.0748

> plot(x ~ r, data=myDF2, type="s", col="blue", # 1位を採用できる確率を表示
+       main=paste("n =", n), ylab="probability")
> ## 理論的に良いとされる r の値 (n が十分大きい場合)
> n/exp(1)

[1] 9.196986

> abline(v=n/exp(1), col="red")

(mc-secretary.r)

```

5.3 補遺

5.3.1 参考文献

この章に関連する参考書としては以下を挙げておく。

- [1] 藤澤洋徳. **確率と統計**. 東京: 朝倉書店, 2006.
- [2] 吉田朋広. **数理統計学**. 東京: 朝倉書店, 2006.
- [3] 竹内啓. **数理統計学**. 東京: 東洋経済, 1963.
- [4] 金明哲. **Rによるデータサイエンス(第2版)**. 東京: 森北出版, 2017.
- [5] U. リゲス (石田基広訳). **Rの基礎とプログラミング技法**. 東京: 丸善出版, 2012.
- [6] 奥村晴彦. **Rで楽しむ統計**. 東京: 共立出版, 2016.
- [7] Larry Wasserman. **All of Statistics**. New York: Springer, 2004.
- [8] Gareth James et al. **An Introduction to Statistical Learning with Applications in R**. New York: Springer, 2013.
- [9] 山本義郎, 藤野友和, 久保田貴文. **Rによるデータマイニング入門**. 東京: オーム社, 2015.