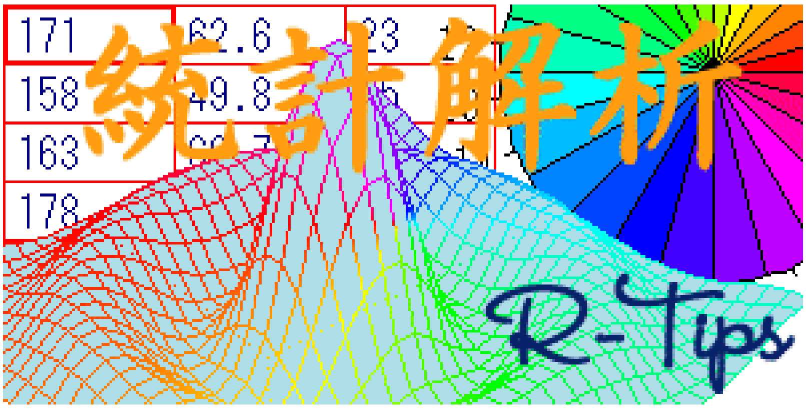


- 統計解析フリーソフト R の備忘録 PDF -

R は有名な統計言語『 S 言語』をオープンソースとして実装し直した統計解析ソフトです．さまざまなプラットフォーム（OS）に対応しており，誰でも自由にダウンロードすることができます．それにもかかわらず，世界中の専門家が開発に携わっており，日々新しい手法・アルゴリズムが付け加えられていますので，機能的にはあの有名な『 S-plus 』を凌駕しているといっても過言ではありません．とにかく計算が速い上にグラフィックも充実しているので数値計算などにも持ってこいです．
このドキュメントは Windows 版 R と Mac OS X 版 R（と一部 Linux 版 R ）でコマンドを調べた足跡です．内容の出所は *http://cse.naro.affrc.go.jp/takezawa/r-tips/r.html* を PDF に直したもの[*1] で，統計解析の内容というよりも R の使い方を中心に解説をしております．

このドキュメントでは頁数などの関係上，統計解析に関する解説を省いております．これは『このパソコン全盛時代において，統計解析を行う際に理論的なことは気にしなくてもよい』ということではありません！私が自信を持って解説することが出来るのは R の使用法のみであって，統計解析に関する理論的な流れは他の先生方に任せた方が良いという意味で載せていないだけであります．もし，R を用いて統計解析をやろうという方で，統計解析の理論的背景が知りたいという方には，以下の参考文献・参考サイトをご覧頂くことをお勧めします．

『 R による統計解析の基礎』 中澤 港 著 (ピアソンエデュケーション)
『工学のためのデータサイエンス入門』 間瀬 茂・神保 雅一・鎌倉 稔成・金藤 浩司 著 (数理工学社)
『The R Book』 岡田 昌史 他 著 (九天社)
『Introductory Statistics with R』 Peter Dalgaard (Springer)
【 R による統計処理 (青木先生の HP) 】：*http://aoki2.si.gunma-u.ac.jp/R/*

いずれも R の解説と統計の解説とが絶妙に融合したものだけを挙げております．統計解析のみを扱った本・解説書・演習本・例解集の名著は書店に溢れておりますが，ここでは 3 冊だけ挙げます．

『数理統計学 (改訂版) 』 稲垣 宣生 著 (裳華房)
『統計学の基礎』 栗栖 忠・濱田 年男・稲垣 宣生 著 (裳華房)
『基本演習 確率統計』 和田 秀三 著 (サイエンス社)

(注意) このドキュメントでは，本文では半角 \ と半角 ¥ を ¥ で，R の命令例・ソースでは半角 \ と半角 ¥ を \ で表記しております．UNIX や LINUX ，Mac OS などでは半角バックスラッシュは \ と表示され，Windows では半角 ¥ で表示されると思いますが，お使いの OS の記号に適宜読み替えて下さい．

[*1] 第 4 版（2005 年 8 月 8 日改訂版）．R のバージョンは 2.1.0 及び 2.1.1 で動作確認．

目次

第 I 部

基本知識篇

第 1 章

インストール篇

1.0　R をインストールせずに試してみる

RWeb (なかまさん) : *http : //r.nakama.ne.jp/Rweb − jp/* で R を試すことが出来る．とりあえず R を試してみたい方に是非お勧めする．

1.1　R のインストール

ここでは，R のバージョン 2.1.0 をインストールする方法を紹介する[*1]．現在の最新版 R のバージョンに読み替えてインストールしていただきたい．

1.1.1　Windows 版 R のインストール

まず，中間さんのサイト：*http : //r.nakama.ne.jp/R − 2.1.0/patched_build/* から実行ファイル rw2010pat.20050420.exe をダウンロードする[*2]．次に，ダウンロードした実行ファイルをダブルクリックしてインストーラーを起動する． Next> をクリックするとライセンスが表示されるので，[I accept the agreement] にチェックを入れてから Next> をクリックする．

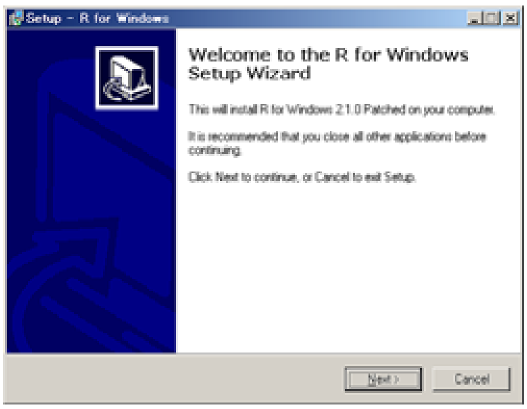


図 1.1　インストーラー

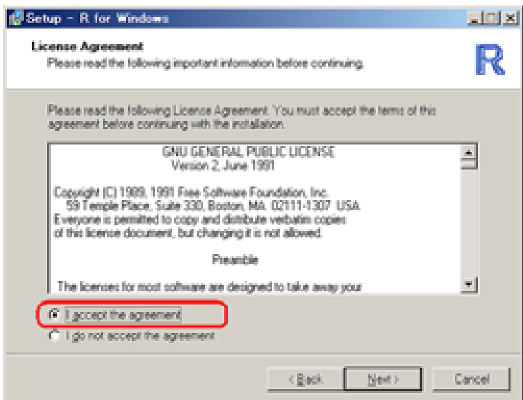


図 1.2　ライセンス

続くウィンドウで Next> をクリックすると，何をインストールするのか選択する画面になる（図 1.4）．ここで，**必ず**プルダウンメニューから [Chinese/Japanese/Korean installation] を選択する．

[*1] R 2.1.1 も同様の手順でインストールすることが出来る．
[*2] パッチ適用済の R 2.1.1 は CRAN（筑波大学）*http : //cran.md.tsukuba.ac.jp/bin/windows/base/rpatched.html* からダウンロードできる．

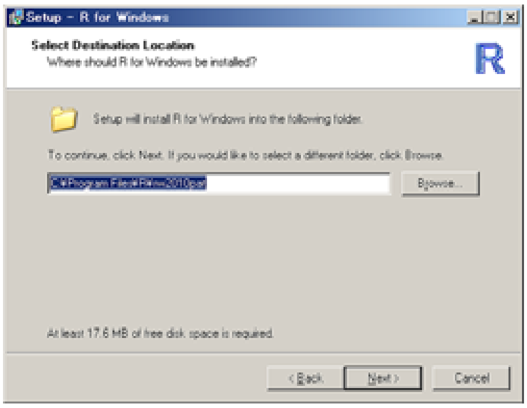


図 1.3　インストール先

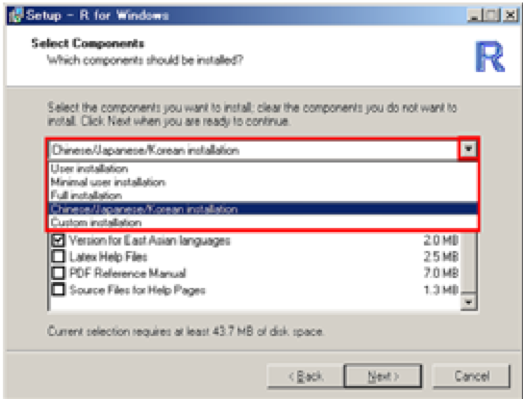


図 1.4　インストールするコンポーネント

[Message Translations] と [Version for East Asian languages] にチェックが入っていることを確認した後（図 1.5），[Next>] をクリックする（スタートメニューの登録名を決める画面）．

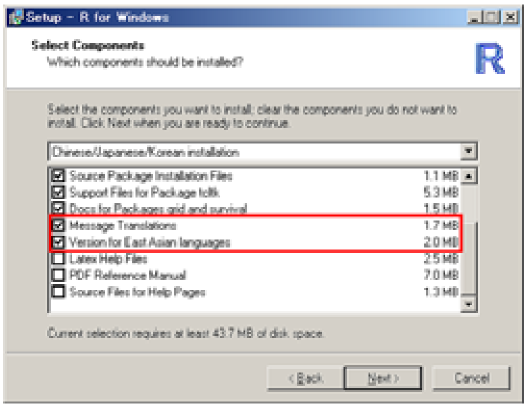


図 1.5　インストールするコンポーネント

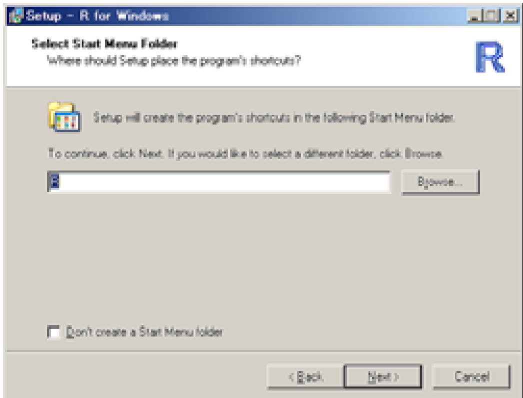


図 1.6　スタートメニューの登録名

図 1.7 は上から順に「デスクトップに R のアイコンを作成する」「クイックランチのアイコンを作成する」「.Rdata という名前のファイルを R のファイルとして関連付ける」「 R にパスを通す」の項目を指定する画面になる．普通はそのまま Next> をクリックする．以上の作業が終了するとインストールが開始される（図 1.8）．

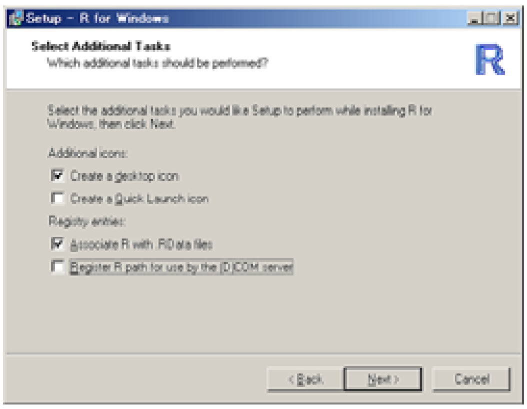


図 1.7　関連付けなど

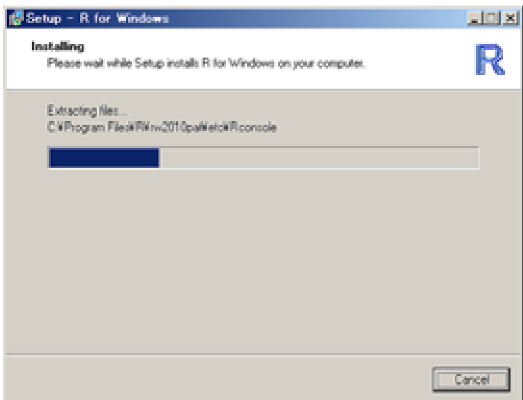


図 1.8　インストール中

インストールが終わると図 1.9 の画面が出る．Finish をクリックしてインストール作業は一旦終了する．最後に，設定ファイルを書き換える作業を行う．R をインストールしたフォルダにある「etc」フォルダ（普通は *C* : ¥*Program Files*¥*R*¥*rw*2010*pat*¥*etc* ，図 1.10 の画面）にある Rconsole と Rdevga を探す．

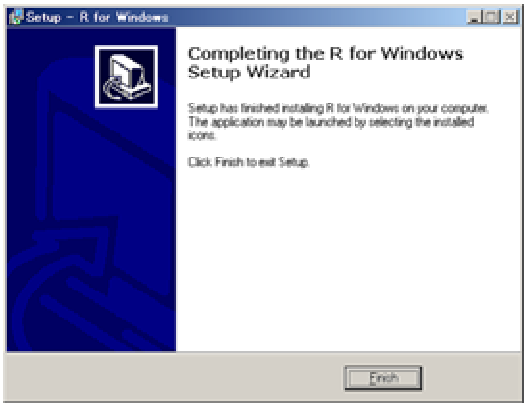


図 1.9　インストール終了

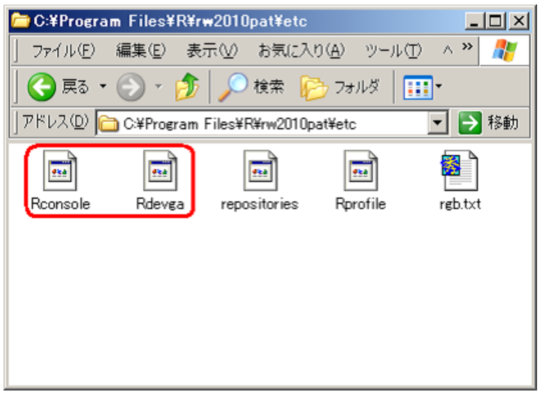


図 1.10　設定ファイルがある場所

Rconsole と Rdevga が見つかったら，これらのファイルをテキストエディタ（メモ帳など）で開き，以下のように書き換える．

`Rconsole`：矢印の左側にある文章を右側の文章に変更する

```
17 行目：font = TT Courier New -> font = TT MS Gothic
18 行目：points = 10            -> points = 14
```

`Rdevga`：12 行目〜15 行目：矢印の左側にある文章を右側の文章に変更する

```
TT Arial : plain       -> TT MS Gothic : plain
TT Arial : bold        -> TT MS Gothic : bold
TT Arial : italic      -> TT MS Gothic : italic
TT Arial : bold&italic -> TT MS Gothic : bold&italic
```

以上でセットアップは終了である．

R の起動

スタートメニューにある [R] の中の [R 2.1.0 patched] をクリックするか，デスクトップに出来た [R 2.1.0 patched] のアイコン をダブルクリックすることで R が起動する．

1.1.2　Mac OS X 版 R のインストール

実行ファイル R-2.1.0.dmg (Mac OS X 10.3 以降) または R-2.0.1.dmg (Mac OS X 10.2) を．The R Project（筑波大学のミラー）：*http : //cran.md.tsukuba.ac.jp/bin/macosx/* からダウンロードする．次に，実行ファイルをダブルクリックするとマウントが行われる．その後，マウントされた R 2.1.0 mpkg をダブルクリックする．

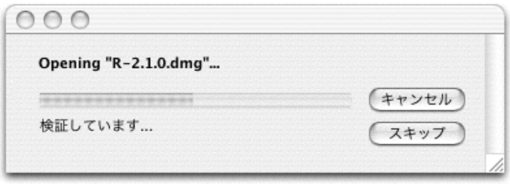


図 1.11　インストーラー

図 1.12　マウント

2 回ほど 続ける をクリックする．

図 1.13　インストール中

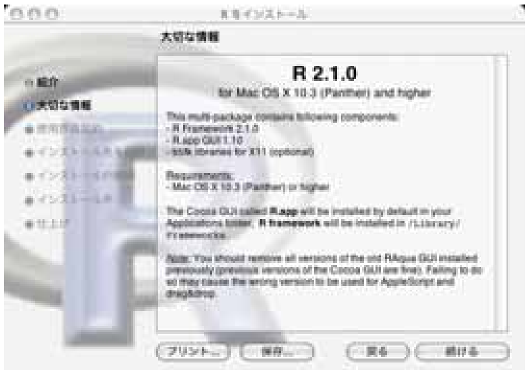


図 1.14　インストール中

すると，ライセンスが表示されるので， Continue をクリックした後，「同意しますか？」という問いに対して Agree をクリックする．

図 1.15　ライセンス

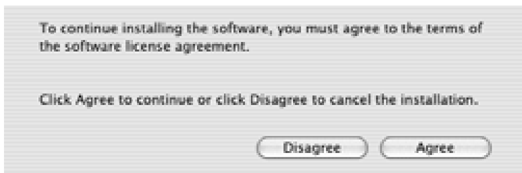


図 1.16　ライセンス

インストールするフォルダを選択する画面になるので，そのまま 続ける をクリックする・その後， インストール または アップグレード のいずれかのボタンが表示されるので，いずれかのボタンをクリックする．

図 1.17　インストール中

図 1.18　インストール中

インストールが終わると図 1.19 の画面が出る．[Finish] をクリックしてインストール作業を終了する．以上でセットアップは終了である．

図 1.19　インストール終了

図 1.20　実行例

R の起動

アプリケーションフォルダに [R] というアプリケーションが出来ているので，これをダブルクリックすることで R が起動する（図 1.20）．

1.1.3　Linux 版 R のインストール

Linux 版 R の詳しいセットアップ方法については RjpWiki:*http://www.okada.jp.org/RWiki/index.php?RjpWiki* を参照されたい．ちなみに，RjpWiki はオープンソースの統計解析システム R に関する日本語による情報交換を目的とした掲示板で，ここでは Wiki とよばれるシステムを採用し，誰でもでも自由にページを追加・編集出来る．「単語検索」から「Linux 版 R のセットアップ」について検索すると良いだろう．

第 2 章

基本知識篇

2.1　最低限必要な知識

2.1.1　起動

R の起動方法は OS によって異なる．

Windows：　デスクトップにある [R 2.1.0 Patched] と書かれたアイコンをクリックするか，スタートメニューのプログラムから [R] というカテゴリの中にある [R 2.1.0 Patched] を選択する．

Mac OS X：　Finder の中のアプリケーションフォルダにある [R] というアイコンをダブルクリックする．

Linux：　Linux といってもいろいろな種類があるが，kterm などのコマンドライン上から R と入力するか，メニューから [Gnu R] を選択することで起動する．

R を起動すると，以下のような画面（ウィンドウ）が表示される．R ではこの入力画面上で主な処理を行う．

```
R : Copyright 2005, The R Foundation for Statistical Computing
Version 2.1.0 Patched (2005-04-20), ISBN 3-900051-07-0

R はフリー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。


>
```

一番下を見ると，> が表示されている．[*1]．これは R がユーザからの入力を待っていることを意味する．この状態で計算式や関数を入力して ↲ キー（ Enter キー）を押せば計算してくれる．

[*1] プロンプト > の記号がお気に召さないならば options(prompt="記号 ") で変更することができる (例：options(prompt=": ")).

```
> 1+2 ↵
[1] 3
> sin(pi/2) ↵
[1] 1
>
```

つまり，R では以下の様な手順を繰り返して作業を行うことになる[*2]．

(1) 我々が式を入力すると，R は入力式を読んで計算し，結果を表示する．
(2) また我々が式を入力すると R は式を読んで計算し・・・(以下くり返し）

そして再び > の記号が現れた．これは「1 + 2 の計算が終了したので，次の計算式を入力してください」と R が要求していることを意味する．この後，新たに計算式を入力して ↵ を押せば R は再び計算処理をしてくれる．

（注意） 上の式を実際に入力する際は > を入力する必要はない．ここでは「1 + 2」だけを入力して Enter を押すだけでよい．

2.1.2　終了

ウインドウの × をクリックしても良いが，以下の 2 通りの方法で R を終了させることが出来る．

```
> q() ↵
> quit() ↵
```

Windows の場合は，メニューの [ファイル] から [終了] を選択しても良い．すると，作業スペースを保存するかどうかを聞くダイアログが表示されるので，はい か いいえ をクリックする．

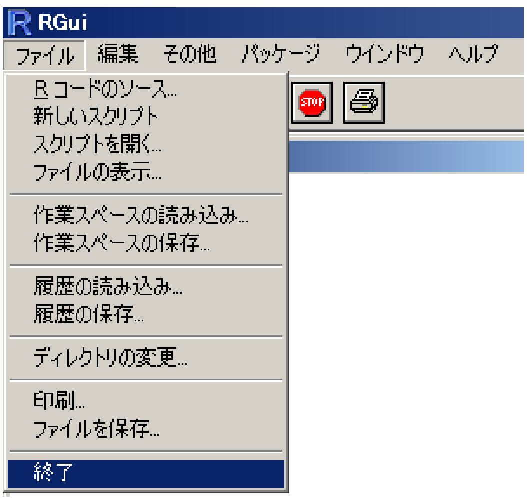


図 2.1　終了

図 2.2　保存ダイアログ

Mac OS X の場合は，メニューの [R] から [R を終了] を選択しても良い．すると，作業スペースを保存するかどうかを聞くダイアログが表示されるので，保存 か 保存しない をクリックする．

[*2] 「1 + 2」が計算されて 3 という結果が得られたが，よく見ると [1] というものが前についている．これは「結果の数字がひとつである」ことを表している．

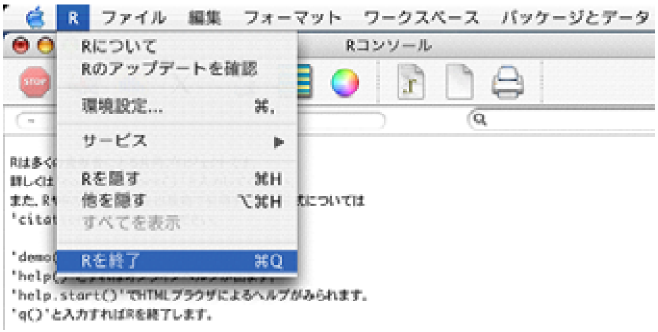


図 2.3　終了

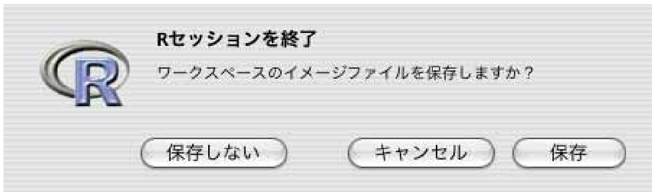


図 2.4　保存ダイアログ

明示的に作業スペースを保存する場合は，関数 save.image(”ファイルのパス”) を実行すればよい．

```
> save.image("C:/filename.RData") ↲
```

2.1.3　簡単な式の計算（演算子と関数）

R では以下の演算子を使って計算が行える．下の表に載っている演算子以外に，“(” や “)” などの括弧を使うことも出来る．

記号	+	^　**	−	%/%	*	%%	/
意味	足し算	累乗	引き算	整数商	掛け算	剰余	割り算

以下に例を示す．

```
> (1 + 2 - 3 * 4) / 5^6 ↲
[1] -0.000576
```

ルートの計算や対数の計算などを行う場合は関数を用いる．R には多数の数学関数が用意されており，例えば $\sqrt{2}$ を計算する場合は sqrt(2) とすればよい．この sqrt() というものが関数で，括弧の中に計算したい数を入れればよい．

```
> sqrt(2) ↲
[1] 1.414213
```

複数の関数が含まれた計算を一度に行うことも出来る．

```
> sqrt(100) + round(100) / log10(100) ↲
[1] 60
```

以下に R で用意されている数学関数を紹介する．表中の x は 2.0 や −0.5 などの実数を表すものとする．

記号	sin(x)	cos(x)	tan(x)	sinh(x)	cosh(x)	tanh(x)
意味	sin	cos	tan	sinh	cosh	tanh
記号	asin(x)	acos(x)	atan(x)	asinh(x)	acosh(x)	atanh(x)
意味	sin の逆関数	cos の逆関数	tan の逆関数	sinh の逆関数	cosh の逆関数	tanh の逆関数
記号	log(x)	log10(x)	log2(x)	log1p(x)	exp(x)	expm1(x)
意味	対数	常用対数	底が 2 の対数	log(1+x)	e^x	exp(x)−1
記号	sqrt(x)	round(x)	trunc(x)	floor(x)	ceiling(x)	signif(x,a)
意味	ルート	四捨五入	整数部分	小数を切捨	小数を切上	x を有効桁 a 桁で丸め

2.1.4　オブジェクトと代入（付値）

R が作ったり操作した実体はオブジェクト (object) と呼ばれる．変数，数の配列，文字列関数，データ，関数その他全てがそれにあたる．オブジェクトは単なる「モノ」であるという理解でよい．「オブジェクトが・・・」 と言われたら「ああ，変数だか数値だか何かがあるんだなあ」と考えてもらえればよく．難しく考える必要は無い．

オブジェクトには代入（付値）によって名前をつけることが出来る[*3]．普通のプログラムで云えば【変数 = 数値 ; 】と，変数に数値を代入する動作に当たり，代入によって名付けられたオブジェクトはその名前で参照することが出来る．以下の例は x に値 2 を代入する例で，代入した名前は後に入力する式の中で自由に使うことが出来る．

```
> x <- 2 ↲
> x ↲
[1] 2
> x + 3 ↲
[1] 5
```

式の中でオブジェクトが使われた場合，それは代入された値を意味する．例えば x <– 2 とした後で x + 3 と入力すれば，これは 2 + 3 と認識される．また，式の中で使われた名前にその式を評価した結果を再度代入することも出来，古い値が捨てられて新しい値が保存される．さらに，変数の値を別の変数にコピーすることも出来る．

```
> x <- 2 ↲
> x <- 3 ↲
> x
[1] 3
> y <- x ↲
```

代入を行う場合は以下の演算子・関数を使うことが出来る．通常は <– を使う[*4]．

代入演算子・代入関数	<–	–>	=	assign(”変数名”, 値)

以下のように丸括弧 () で囲むと，代入と表示を同時に行なう．

```
> (x <- 1:4) ↲
[1] 1 2 3 4
```

ただし，以下の名前は R の処理系によって予約されているので，オブジェクトの名前として用いることは出来ない[*5]

break　else　for　function　if　in　next　repeat　return　while　TRUE　FALSE

2.1.5　以前に計算した式を再び呼び出す（履歴）

前に使った命令を再度使う際，キーボードの上矢印 ↑ や下矢印 ↓ で前の命令を表示することが出来る．ここで ↲ を押せば，再度同じ計算を実行してくれる． ↑ を何回も押すことで，過去に入力した計算式が遡って表示される[*6]．

[*3] 変数名はローマ字や数字を使うことが出来，この 2 つを組み合わせることも出来るが，変数名の先頭は数字にしてはいけない（エラーが出る）．例えば，A3 という変数例は許されるが，3A という変数名は許されない．

[*4] 「y <– x <– 10」と，一度に複数のオブジェクトに値を代入することも出来る．

[*5] 関数 objects() を使うことで，現在どんなオブジェクトがあるかを調べることが出来る．また，前に定義したオブジェクトを消す場合（例えば x の定義や x に入っている値を消したい場合）は関数 rm() を用いる．さらに，全てのオブジェクトを消す場合は rm(list=ls(all=TRUE)) とすればよい．

[*6] 他にも，今までに入力した命令を一覧で表示する関数 history() がある．

2.1.6　R 用エディタと本文の記載方法について

R では 1 つの命令が数行にわたる場合が多い．このような場合，R の入力画面に直接入力すると，間違ったときにもう一度始めから入力し直さなければならなくなるので不便である．1 つの命令が数行にわたる命令を記述する際，いきなり R の入力画面に直接入力するのではなく，適当なエディタに命令を書いておいて，まとめて実行するのがよい．

Windows の場合

メニューの [ファイル] から [新しいスクリプト] を選択すると，R 用のエディタが立ち上がる．そこに命令を記述することが出来る．命令を実行する場合，実行する命令をマウスなどで範囲選択し・・・

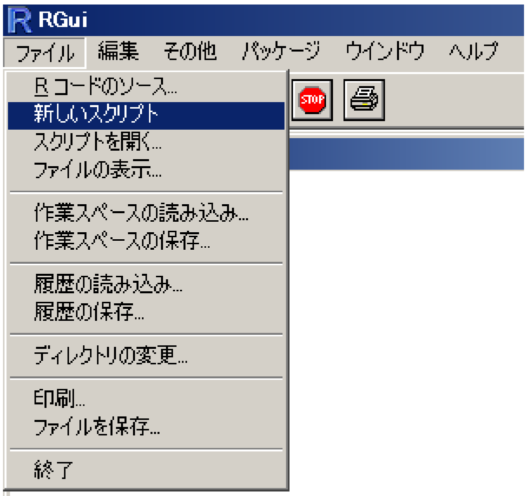


図 2.5　新しいスクリプト

図 2.6　命令を記述して範囲を選択

[編集] の [カーソル行または選択中の R コードを実行] を選択すると，命令が実行されて結果が表示される．アイコン をクリックするか，Ctrl+R や F5 でも同様のことが行える[7]．

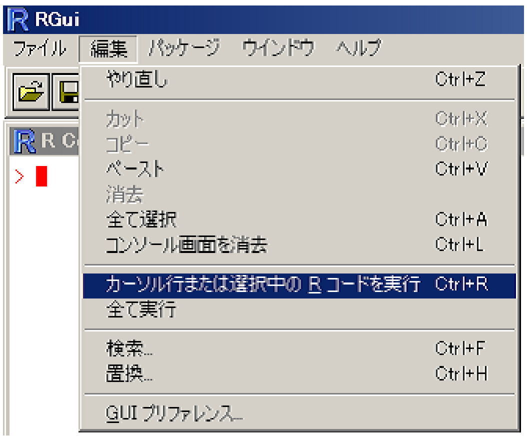


図 2.7　範囲を選択して実行

図 2.8　実行結果

[7] このショートカットキーは実に便利である．例えば，R 用のエディタ記述した命令を全て実行する場合，Ctrl+A を行ってから Ctrl+R とするのが速い．

Mac OS X の場合

メニューの [ファイル] から [新規作成] を選択すると，R 用のエディタが立ち上がる．そこに命令を記述することが出来る．命令を実行する場合，実行する命令をマウスなどで範囲選択し・・・

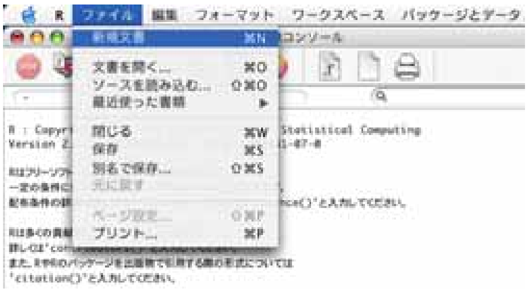


図 2.9　新しいスクリプト

図 2.10　範囲を選択

[編集] の [実行] を選択すると，命令が実行されて結果が表示される．Command+↲ でも同様のことが行える[*8]．

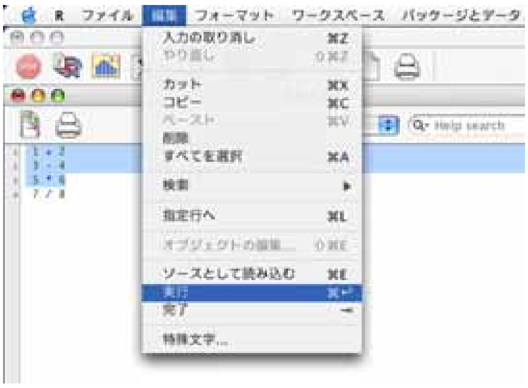


図 2.11　実行

図 2.12　実行結果

Linux の場合

vi や Emacs などに命令を書いておき，まとめてコピーして R のコンソールにペーストする方法が手っ取り早い．また，Emacs や XEmacs から R（などの統計解析アプリケーション）を使うことを目的としたエディタ ESS (Emacs Speakes Statistics)[*9] がある．ESS がどれだけ便利であるかは，ESS のホームページや RjpWiki，「The R Book」の 55 頁～ 56 頁などを参照されたい．

本文の記載方法についての注意

ここでは，R 用エディタを用いて R の命令を記述することを念頭に置く．そこで，R の命令は「命令部分：> や + 付きで記載」「実行結果部分：そのまま記載」と記述し，簡単のため ↲ は省略することにする．

```
> (1 + 2 - 3 * 4) / 5^6
[1] -0.000576
```

[*8] このショートカットキーは実に便利である．例えば，R 用のエディタ記述した命令を全て実行する場合，Command+A を行ってから Command+↲ とするのが速い．

[*9] *http://stat.ethz.ch/ESS/*

さて，入力した計算式のメモを書いておきたい場合がある．そのようなときは # 以下にコメントを書くことが出来る．# 以下の文字列は無視されるので，好きなことを好きなだけ書くことが出来[*10]，コメントをつけても実行結果に変わりはない．前述の命令にコメントをつけた例を挙げる．

```
> (1 + 2 - 3 * 4) / 5^6        # 足して引いて掛けて割って・・・
[1] -0.000576
```

以降，本書では入力式の説明をコメントで行う場面が多数出てくるが，コメントの文章は入力する必要はない（念のため）．先ほどの例であれば，「(1 + 2 − 3 ∗ 4) / 5ˆ6」とだけ入力すればよい．

2.1.7　作業ディレクトリの変更

ファイルからデータやプログラムを読み込んだり，ファイルにデータを書き出したりする場所を作業ディレクトリという．起動時はホームディレクトリ (R の実行ファイルがある場所) が作業ディレクトリとなっているが，以下のような命令を与える[*11] と，これ以後，指定したディレクトリに指定した作業ディレクトリにデータがセーブされたり，R 用エディタなどが保存されるようになる[*12].

```
> setwd("c:/usr")              # 作業ディレクトリを変更する
> getwd()                      # 現在の作業ディレクトリを確認する
[1] "c:/usr"
```

Windows の場合，メニューの [ファイル] から [ディレクトリの変更...] を選択してから，ディレクトリを変更することが出来る．

図 2.13　ディレクトリの変更

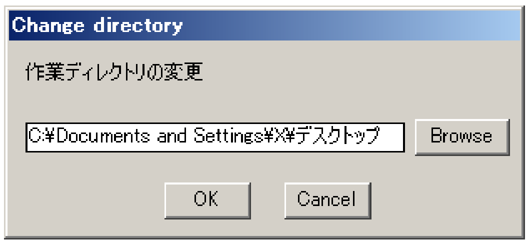


図 2.14　ディレクトリの変更

Mac OS X の場合，メニューの [その他] から [作業ディレクトリの変更...] を選択すれば，新しい作業ディレクトリを選択することが出来る．

[*10] ただし英語版の R で日本語コメントを書いた場合は文字化けしてしまうので注意が要る．
[*11] Windows では ¥を指定しては駄目．例えば，setwd(”c:¥usr”) とするとエラーが出る．setwd(”c:/usr”) とすること．
[*12] 関数 dir() で作業ディレクトリ内のファイルが表示される．

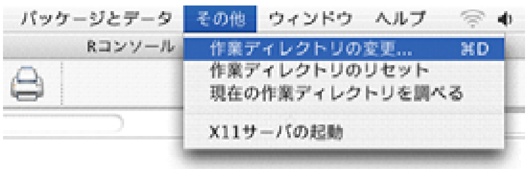


図 2.15　作業ディレクトリの変更

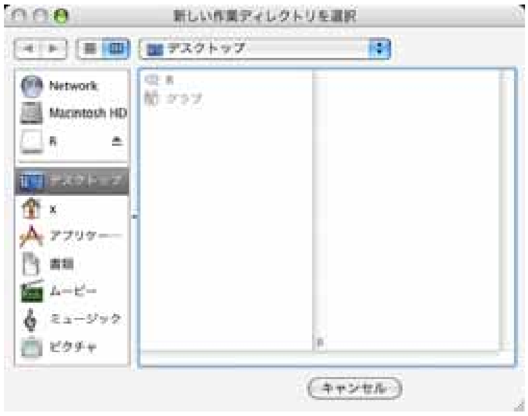


図 2.16　新しい作業ディレクトリの選択

2.1.8　ヘルプを見る

プログラムを書いていて例えば，関数 solve() をどう使うか分からなくなった場合は，以下の 2 通りの方法で solve に関するヘルプを見ることが出来る[*13].

```
> help(solve)
> ?solve
```

関数に限らず R の文法などの総合的な解説やデータセットなどの解説も同じように見ることができる．例えば文法の解説は以下のようにすると表示される．

```
> ?Syntax
```

for などの予約語や，[などの特殊記号の解説を見るには ”” で囲む必要があるので注意が要る (> ?”for” でも可)

```
> help("for")
```

関数名ではないが「方程式を解く（solve）機能を持った関数って何？」という場合は，関数 help.search(”機能を表す単語”) で該当する関数が表示される[*14].

```
> help.search("solve")
```

2.1.9　落穂ひろい

ここまでで紹介し切れなかった事項を落穂ひろいする．面倒ならば読み飛ばして頂いて構わない．

(a) R は大文字と小文字を区別する．従って x と X は異なるものとして R は認識する．

(b) 式を入力する際に，sin などの名前の中や後に出てくる文字列の中を除いて，空白はいくつあっても構わない．例えば以下の 2 つの命令ははどちらも同じ結果を返す．

```
> 1          +          2
[1] 3
> 1 + 2
[1] 3
```

[*13] 関数の完全な解説は必要無いが，引数のみを知りたいという場合は args(solve) とすればよい．

[*14] インターネットに接続されている場合は，関数 RSiteSearch(”キーワード”) で R-help mailing list のアーカイブ，R マニュアル，R のヘルプ頁からキーワードを探してくれる．

(c) セミコロン ; で区切ることで，複数の式をまとめて一行に書くことも出来る．

```
> 1+2;   3-4
[1] 3
[1] -1
```

(d) 括弧が閉じていない場合や演算式が終了していない場合など，式が未完成のままで改行を入力してしまうと，R は通常のプロンプト > の代わりに式が継続していることを示す + を表示する．この場合，引き続いて続きの式を入力することが出来る．

```
> 1 +
+ 2
[1] 3
```

2 行目先頭の + は「式が継続していることを示す +」であり，「1 + +2」という入力ではない．> と同様，「式が継続していることを示す +」は実際には入力する必要は無く，「1 + 2」とだけ入力すればよい．また，計算式の入力が途中の状態で，その計算式の入力をやめたい時は Esc キーを入力すればよい．

(注) S-PLUS や S 言語では，文字列を示す ” や ’ が閉じていない場合は文字列が継続していることが一目でわかるように + の代わりに“ Continue string: + ” が使われる．どちらもこの状態から式の続きを入力することが出来，式が完成した時点で評価が行なわれる．しかし R ではエラーが出る場合があるので注意．

(e) R は数値の他に論理値やベクトル，文字列などを扱うことができる．後の章で詳しく解説するが，例えば x というベクトルを定義したければ以下のようにすればよい．

```
> x <- c(1,2)
```

本当に x にベクトル (1, 2) が入っているかを確かめてみる．

```
> x
[1] 1 2
```

(f) 丸括弧 (を用いると代入と表示を同時に行なうことは既に扱ったが，以下のような場面でも同じ働きをする．

```
> x <- c(1, 2)                        # 代入のみで結果の表示はない
> (x <- c(1, 2))                      # 丸括弧で囲むと、代入した結果を表示
[1] 1 2
> f <- function(x) y <- 2*x
> f( c(1, 2) )                        # 何も表示されない
> f <- function(x) (y <- 2*x)
> f( c(1, 2) )                        # 丸括弧で囲むと代入した上で表示される
[1] 2 4
```

(g) R のヘルプは英語なので日本人には読みにくい場合がある．手っ取り早く使い方が見たいという場合，例えば関数 solve() の使い方が知りたい場合は以下のようにすれば例を出力してくれる．

```
> example(solve)
```

ただし，グラフ出力がある場合で，グラフをいくつか含む場合は，単に example() をやってしまうと数枚のグラフが一瞬で表示されきってしまう場合がある．動体視力に自信のない方は，以下を定義しておくだけでグラフの描画ごとに逐一静止してくれる．

```
> par(ask = TRUE)
```

(h) Windows 版 R のスクリプトに命令を書いている場合で，書いたプログラムを保存する場合は，メニューの [ファイル] から [保存] を選択すればよい（図 2.17）．また，保存したファイルを開く場合は，まず，メニューの [ファイル] から [スクリプトを開く] を選択する（図 2.18）．

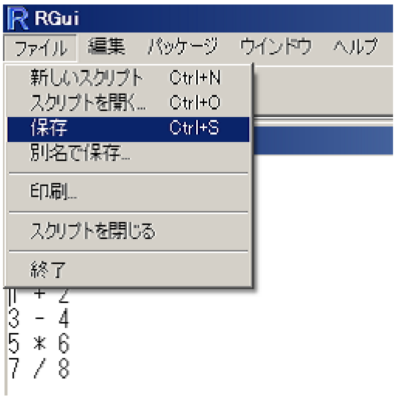


図 2.17　スクリプトの保存

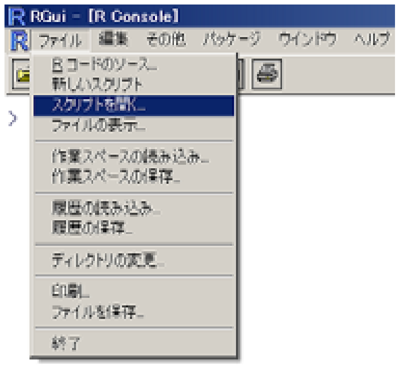


図 2.18　スクリプトを開く

すると，ファイルを選択するダイアログが開くので，拡張子が「.r」か「.q」かそれ以外なのかを選択した後，目的のファイルを選択すればよい．

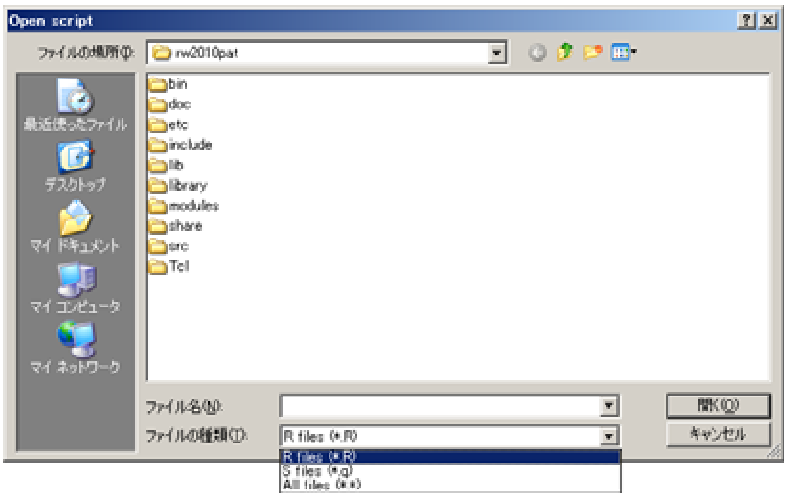


図 2.19　ファイルを開くダイアログ

Mac OS X 版 R のテキストファイル（文書）に命令を書いている場合で，書いたプログラムを保存する場合は，メニューの [ファイル] から [保存] を選択すればよい（図 2.20）．また，保存したファイルを開く場合は，まず，メニューの [ファイル] から [文書を開く] を選択する（図 2.21）．

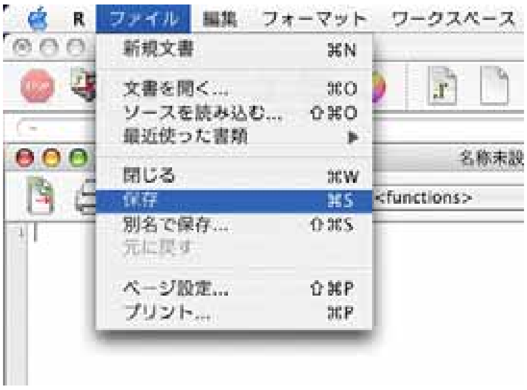


図 2.20　文書の保存

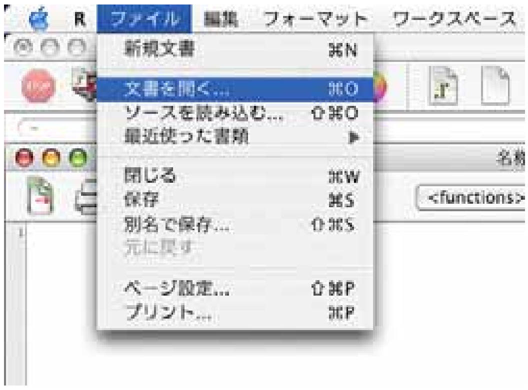


図 2.21　文書を開く

すると，ファイルを選択するダイアログが開くので，目的のファイルを選択すればよい．

図 2.22　ファイルを開くダイアログ

もし，関数の定義を "source.txt" に書いている場合で，関数の定義のみをファイルから読み込みたい場合は以下のようにする．

```
> source("C:/source.txt")
```

上の例のように，ファイル名だけでなくファイルのある場所（パス）も書くのが確実であるが，作業ディレクトリのフルパスを指定しない場合は現在の作業ディレクトリで data.txt が探されることになる（作業ディレクトリの参照・変更方法は 22 頁を参照のこと）．初期状態では R の実行ファイルなどが入っているディレクトリ（Windows ならば C:¥Program Files¥R¥rw2010pat ，Mac OS X ならば /Users/（ユーザー名））になっている[*15]．

[*15] 前述の通り，関数の定義を "source.txt" に書き source("source.txt") 関数で読み込むことが出来るが，R の初期設定ファイル .Rprofile に source("source.txt") を入れておけば，いつも自動的に読み込まれる．また， R セッション中に関数定義を save() 関数で Rdata ファイルに保存しておくことも出来る．これは関数 attach() で定義ファイルを読み込むことが出来る．

```
> myfunc <- function(x) x^2
> save(myfunc, file="source.Rdata")
 ... 一旦終了 ...
> attach("source.Rdata")
```

2.2　パッケージ・ライブラリ

R は関数とデータを機能別に分類してパッケージ（ライブラリ）という形にまとめている．どのようなパッケージが用意されているかは，関数 library() を実行することで知ることが出来る．

```
> library()
```

標準では以下のパッケージが利用出来る．

パッケージ名	説明
base	R の基本パッケージ
boot	ブートストラップに関する R（S-Plus）の関数パッケージ
class	Classification に関する関数パッケージ
cluster	クラスター分析用関数パッケージ
foreign	R 以外の統計ソフト (SAS など) で作成されたデータファイルからデータを読むパッケージ
grid	グリッド・グラフィックス・パッケージ
KernSmooth	カーネル関数による密度推定用関数パッケージ
lattice	ラティス・グラフィックス関数パッケージ
MASS	『Modern Applied Statistics with S』に出てくるデータセットと関数パッケージ
methods	R のオブジェクト用に定義されたメソッドやクラス, プログラミング・ツールのパッケージ.
mgcv	多変量平滑化パラメータ推定 (GCV,GAM) パッケージ
nlme	線形混合効果モデル及び非線形混合効果モデル用のパッケージ
nnet	フィード・フォワード・ニューラル・ネットおよび多項式の対数線形モデルパッケージ
rpart	再帰アルゴリズムによる分類と回帰樹（regression trees）に関するパッケージ
spatial	クリギングと点パターン解析パッケージ
splines	スプライン回帰パッケージ
stats	R の統計パッケージ
stats4	S4 クラスの統計関数
survival	ペナライズ尤度を含む生存時間解析パッケージ
tcltk	Tcl/Tk へのインターフェースパッケージ
tools	パッケージ開発と管理用ツールパッケージ
utils	R のユーティリティパッケージ

2.2.1　パッケージの説明

標準パッケージ以外にも 500 以上の拡張パッケージが用意されており（2005 年 6 月現在），自由にインストールして使用することが出来る．各パッケージの詳しい説明は以下のコマンドで表示される．

```
> library(help="<パッケージ名>")
```

標準で検索リストに登録されているパッケージ (例えば base) はそのまま使うことが出来る．それ以外のライブラリを使用するには，パッケージ名を引数とする library(パッケージ名) を実行すれば良い．

例 1：現在読み込んでいるパッケージの一覧を見る

```
> search()
```

例 2：パッケージ rgl を呼び出す

```
> library(rgl)
```

例 3：呼び出したパッケージ rgl を使ってみる

```
> example(rgl.surface)
>
> n <- 100
> rgl.clear()
> rgl.bg(color=c("white", "black"))
> rgl.spheres(rnorm(n), rnorm(n), rnorm(n), radius=0.2, color=rainbow(10))
> rgl.bbox(color="#333377")
```

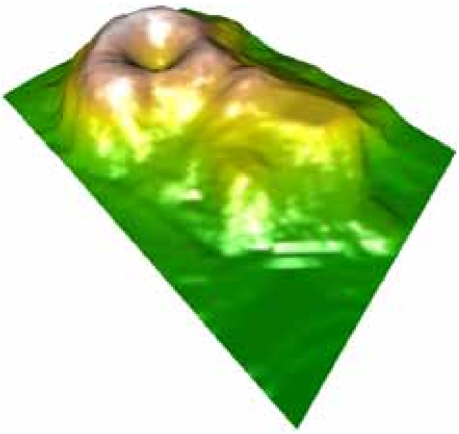


図 2.23　火山（データ volcano）

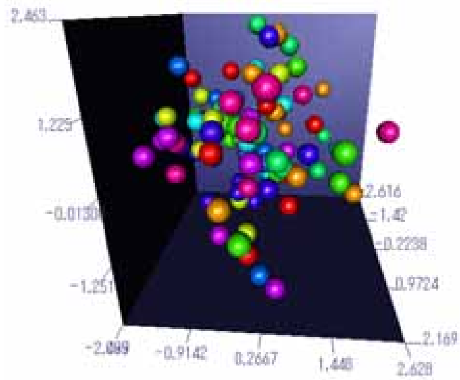


図 2.24　三次元散布図

参考までに，パッケージ rgl の関数一覧を紹介する．

関数	機能	関数	機能
rgl.open()	デバイスの生成	rgl.set()	デバイスのセット
rgl.cur()	デバイス番号の表示	rgl.close()	デバイスを閉じる
rgl.clear()	図の消去	rgl.pop()	現デバイスの消去
rgl.quit()	デバイスの終了	rgl.viewpoint()	視点のセット
rgl.light()	光源のセット	rgl.bg()	背景色の設定
rgl.points()	点のプロット	rgl.sprites()	スプライトのプロット
rgl.lines()	線のプロット	rgl.triangles()	三角のプロット
rgl.quads()	多角形のプロット	rgl.texts()	文字のプロット
rgl.surface()	三次元プロット	rgl.spheres()	球のプロット
rgl.snapshot()	図の保存	rgl.bbox()	箱のプロット

例 4：読み込んだパッケージ rgl の詳細情報を見る

```
> library(help="rgl")
> help(package="rgl")
```

例 5：パッケージ rgl を unload する場合

```
> detach("package:rgl")
```

CRAN 上のパッケージから自分の環境にあるパッケージをアップデートする場合は，関数 update.packages() を用いればよい．

（注意） Windows 版 R では，図形をマウスでクリックしようとすると，R のメインウインドウが rgl のウインドウを隠してしまうことがある．この場合，rgl の ウインドウを最大化するか，R のメインウインドウと重ならないように rgl の ウインドウを配置する必要がある．

（参考） パッケージにある関数やデータセットに関するヘルプを表示する場合は以下のようにする．

```
> help.search("solve")          # solve という機能がある関数の情報を得る
> help(package="MASS")          # パッケージ ’MASS’ に関する関数の一覧
> base::log                     # base パッケージ中の関数 log の情報を得る
> base::"+"                     # base パッケージ中の演算子 "+" の情報を得る
> grid::grid.newpage()          # grid パッケージを自動ロードするのでエラーにならない
> data()                        # 使用可能なデータセットを表示
> data(package = "base")        # パッケージ base で使用可能なデータセットを表示
> data(USArrests, "VADeaths") # データセット ’USArrests’ と ’VADeaths’ をロード
> help(USArrests)               # データセット ’USArrests’ に関するヘルプ
```

2.2.2 拡張パッケージをインストール

また，R-Project の CRAN から拡張パッケージをインストールすることも出来る．日本語によるパッケージの簡単な説明は RjpWiki の CRAN パッケージリストから見ることが出来る．CRAN から特定のパッケージをインストールする場合，まず CRAN の URL を関数 CRAN.packages() で設定した後，関数 install.packages() でパッケージを指定する．以下では例としてパッケージ xtable をインストールしている．

```
> options(CRAN="http://cran.r-project.org")
> install.packages("xtable")
```

圧縮ファイルからパッケージをインストールする場合は，関数 install.packages() でファイルのパスを指定してインストールする．以下では例としてパッケージ xtable_1.2-3.zip をインストールしている．

```
> install.packages("c:/xtable_1.2-3.zip", CRAN = NULL)
```

Windows 版 R の場合

Windows 版 R ならば，GUI のメニューから簡単にパッケージをロードしたり，インストールしたりすることが出来る．まず，メニューの [パッケージ] から [パッケージのインストール] を選択する[*16]．次に，CRAN のミラーサイトを選択する[*17]．通常は日本のミラーサイトを選択すればよい．

[*16] 関数 installed.packages() と同じ機能．

[*17] 関数 CRAN.packages() と同じ機能．

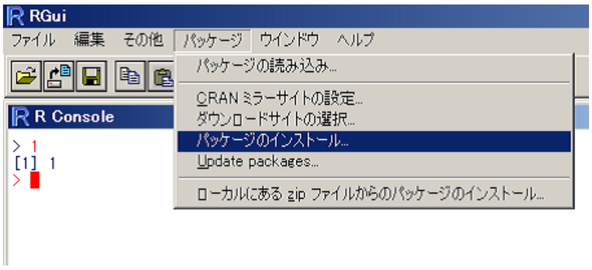


図 2.25　パッケージのインストール

図 2.26　CRAN のミラーサイトを選択

すると，どのパッケージをインストールするかを選択するダイアログが表示されるので，選択してから OK をクリックする（図 2.27）．すると，ダウンロードが開始され，パッケージがインストールされる．パッケージを呼び出す場合，まずメニューの [パッケージ] から [パッケージの読み込み] を選択する（図 2.28）．すると，読み込むパッケージを選択するダイアログが表示されるので，選択してから OK をクリックすればよい（図 2.29）．

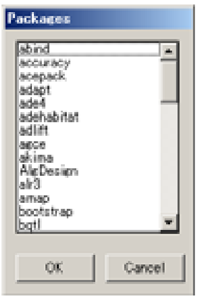


図 2.27　パッケージの選択

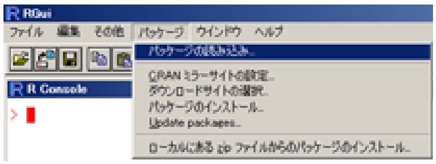


図 2.28　パッケージの読み込み

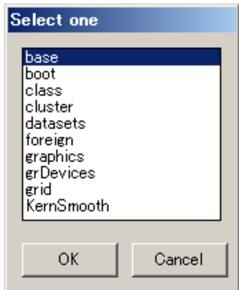


図 2.29　パッケージの選択

企業の Windows マシンで R を使っている場合で，プロキシサーバでインターネットにアクセスしている場合は，R のショートカットのプロパティの「リンク先」の後ろに「--internet2」を追加する必要がある．（例：”C:¥Program Files¥R¥rw2010¥bin¥Rgui.exe” --internet2）

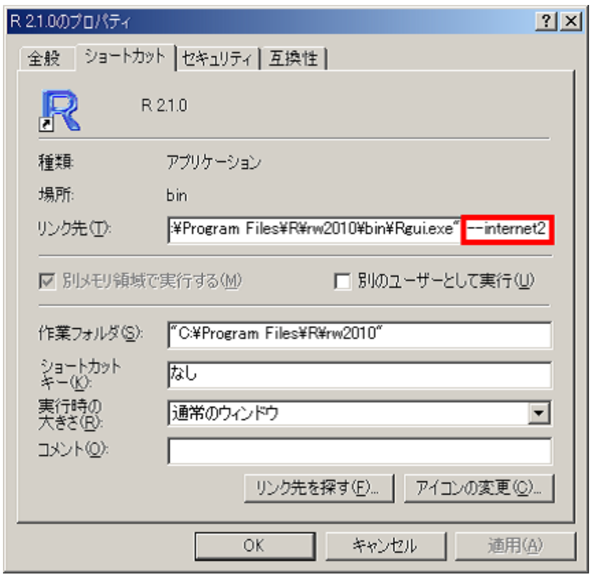


図 2.30　パッケージの読み込み

Mac OS X 版 R の場合

Mac OS X 版 R の場合でも，GUI のメニューから簡単にパッケージをロードしたり，インストールしたりすることが出来る．まず，メニューの [パッケージとデータ] から [パッケージインストーラ] を選択する．次に，一覧を取得をクリックする．

図 2.31　パッケージのインストール

図 2.32　CRAN のミラーサイトを選択

すると，どのパッケージをインストールするかを選択するダイアログが表示されるので，選択してから OK をクリックする．すると，ダウンロードが開始され，パッケージがインストールされる．

図 2.33　パッケージの選択

図 2.34　ダウンロード

パッケージを呼び出す場合，まずメニューの [パッケージとデータ] から [パッケージマネージャ] を選択する．すると，読み込むパッケージを選択するダイアログが表示されるので，目的のパッケージのチェックボックスにチェックを入れ，一覧の更新をクリックすればよい．

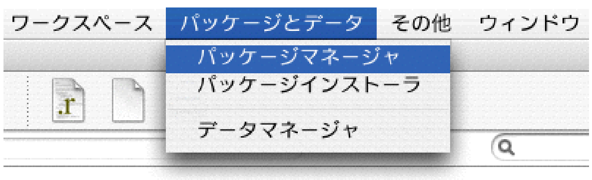


図 2.35　パッケージの読み込み

第 II 部

データ構造篇

第 3 章

データの型とオブジェクトの表示

3.1　データの型

R には「データの型」という概念があり，実数や複素数，文字列などをそれぞれ別の型として区別している．

3.1.1　空 → NULL

「何もない・空っぽ」というものを，R では NULL と表現する．0 と混同しがちだが，NULL は 『何もない・0 すらない』という意味である．後に出てくるが，配列の初期値や names 属性などの属性を取り除くのに使われたり，for などの反復演算を行うときの配列やデータリストの初期値として用いられる．

```
> x <- c()
NULL
```

3.1.2　欠損値・不定データ → NA，非数 → NaN，無限大 → Inf

欠損値 (Not Available) は NA ，非数 (Not a Number) は NaN で表される．

```
> NA                  # データの欠損を表す
[1] NA
> 0/0                 # 数では表せないことを表す
[1] NaN
> 1/0                 # 無限大になる場合は NaN とは別に Inf というものがある
[1] Inf
```

3.1.3　実数 → numeric

例えば，1e10 は 『$1 \times 10\hat{}(10) = 10000000000$』 を表す．また，円周率 π を表す pi も勿論実数である．

```
> 123
[1] 123
> 1e10                     # 10^(10) と同じ
[1] 1e+10
```

3.1.4　複素数 → complex

1 と 1+0i は違う．また，虚数 $1+i$ を表すときは 1+1i と表記すること．1+i とすると i はオブジェクト（変数など）と認識される[*1].

```
> 1 + 1i
[1] 1+1i
> 1 + 0i      # 1 とは異なる
[1] 1+0i
```

3.1.5　文字列 → character

1 文字ではなく，引用符 ”” で囲んだ文字列が単位である[*2]．文字列中にダブルクオートまたはシングルクオートを含めるには，これらの文字の前にバックスラッシュ ¥を置けばよい.

```
> "abc"
[1] "abc"
> ""                          # ダブルクオート " が 2 つあるという意味ではないので注意
[1] ""                        # 中身は " と " の中にあるもの（ここでは何も無いことを表す）
```

3.1.6　論理値 → logical

論理値は TRUE (真) と FALSE (偽) の値をとる．これらはそれぞれ T，F と略記することも出来る[*3]．TRUE や FALSE は文字列ではないので注意.

```
> T
[1] TRUE
> TRUE
[1] TRUE
```

[*1] 実数から複素数に自動的に型変換がなされることはない．以下の例より，-2 と -2+0i は異なることが分かる.

```
> log(-2)
[1] NaN
警告メッセージ：
計算結果が NaN になりました in: log(x)
> log(-2+0i)
[1] 0.6931472+3.141593i
```

[*2] ’ ’ で囲んでも良い．ただし ¥は例外.

[*3] T と F は，それぞれ TRUE と FALSE が代入されているだけである．よって，T や F に TRUE や FALSE 以外の値を代入することも出来る（決してお勧めはしない）.

3.2　オブジェクトの表示

オブジェクト名（変数名）だけを入力してもオブジェクトの中身は表示されるが，以下に紹介する関数を用いることで，出力形式をカスタマイズすることが出来る．

3.2.1　オブジェクトを表示する：print()

オブジェクトを表示する基本的な関数は print() である[*4]．

```
> x <- "one"
> print(x)            # "" ありで出力
[1] "one"
> print(x, quote=F)   # "" なしで出力
[1] one
```

3.2.2　文字列を表示する：cat()

文字列を表示する基本的な関数は cat() である．関数 print() で文字列を表示すると前後にダブルクオート ”” がつけられるが，cat() で表示すればダブルクオート ”” はつかない[*5]．文字列中で ¥¥，¥t，¥n，¥f を用いると，それぞれ ¥マーク，タブ，改行，改ページ文字を出力する[*6]．さらに ¥で始まる 3 桁の 8 進数を書けば，文字列中に文字コードを直接記述することも出来る．

```
> cat("cd C:\\Document and Settings\\usr\\ \n")  # 作業ディレクトリ指定の際に有用
cd C:\Document and Settings\usr
```

cat() は最後に改行を付けないので，改行したければ文字列中に改行記号（¥n）を含める必要がある．

3.2.3　書式付きでオブジェクトを表示する：sprintf()

書式付きでオブジェクトを表示するには関数 sprintf() を使う[*7]．

命令	機能
sprintf(”%5.1f”, 実数)	実数を「■■■．■」(整数部分 3 桁，小数点 1 つ，小数部分 1 桁）で表記する．小数部分がはみ出た場合は丸められ，整数部分が足りない場合はスペースで補われる．
sprintf(”%f”, 実数)	f は小数点以下の桁数を指定する．デフォルトは 6.
sprintf(”%−10f”, 実数)	左詰にする．余った部分は空白で埋める．% と f の間に + や 0 などの文字を入れることも出来る．
sprintf(”%3d”, as.integer(整数))	整数部分 3 桁で表記する．整数部分が足りない場合はスペースで補われる．「%3i」も同様.
sprintf(”%e”, 実数)	指数表示（浮動小数点表記）する．亜種として ”g” や ”G” が指定できる．
sprintf(”%s”, as.character(文字))	数値を文字として出力する場合は as.character() で変換する．

[*4] 関数 page() で別ウインドウにオブジェクトの値を表示することも出来る．これは長いデータなどを表示する場合に有用である．
[*5] 文字列でないオブジェクトは文字列に強制変換される．
[*6] 使っている OS によって，¥マークを適宜 \ マークに読み替えていただきたい．ただし ¥単独で用いるとエラーが起きるので注意．
[*7] 有効桁数を決める関数は，他に formatC() などがある．

フォーマットを ”%5.0f” とした場合は，実数を整数部分 5 桁で表記する．整数部分が足りない場合はスペースで補われ，小数部分がある場合は丸められる．

```
> sprintf("%5.0f", 1234.5678)
[1] " 1235"
```

3.2.4 オブジェクトの内容を情報付きで表示する：str()

オブジェクトの内容を情報付きで簡潔に表示する場合は関数 str() を用いる．データの要約が欲しい場合は関数 summary() を用いる．

```
> str(x)
 chr "12345"
> summary(x)
   Length     Class      Mode
        1 character character
```

3.2.5 オブジェクトにコメントを残す：comment()

オブジェクトに備忘録的にコメントを残しておく場合は関数 comment() を用いる．付け加えたコメントは普通の計算時には表示されない．コメントを表示する場合は関数 comment() や str() を使う．

```
> x <- "Washi"
> comment(x) <- "my handle name"
> x
[1] "Washi"
> comment(x)
[1] "my handle name"
```

3.2.6 出力をファイルに送る：sink()

通常はコマンドライン上に表示される R の出力をファイルに送る場合は関数 sink(”ファイルのパス”) を実行する．似たような関数に dump()，dget()，dput()，capture.output() がある．

```
> sink("output.txt")  # 作業ディレクトリの中に output.txt が出来る．
```

出力をファイルに送るのを解除するには以下の様に入力すればよい．

```
> sink()
```

3.3 オプション

関数 options() でさまざまなオプションを変更できる．options() の引数には整数値か論理値を指定する[*8].

引数	機能
continue	プロンプト記号を設定する（デフォルトは “+ ”).
device	デフォルトで使うデバイス ('x11', 'windows', 'gtk') を設定する.
digits	数値を出力する際の表示桁数を設定する（デフォルトは 7).
expressions	評価する入れ子の数の限界を設定する（デフォルトは 500).
prompt	プロンプト記号を設定する（デフォルトは “> ”).
scipen	指数表現にするか否かの閾値を設定する（デフォルトは 0）
show.error.messages	エラーメッセージを出力するか否かを設定する（デフォルトは TRUE）. FALSE にするとエラーメッセージが出なくなる.
timeout	インターネット接続のタイムアウト時間を設定する（デフォルトは 60 秒).
warn	警告メッセージの取り扱いを設定する（デフォルトは 0). 負の値の場合は警告がすべて無視され，1 の場合は警告が生じるとともに警告が出力，2 以上の場合は警告はすべてエラーに変えられる.
width	ライン上の文字の数を設定する（デフォルトは 80).
CRAN	CRAN の URL を指定する（デフォルトは $http://cran.r-project.org$).

（例 1）　最大表示桁数は 7 であるが，digits (整数) を変更することで最大桁数を変えることが出来る.

```
> options(digits=10)    # 表示桁数を 10 桁に変える
```

（例 2）　例えば，10000 は そのまま表現され，100000 は指数表現 1e5 と表現しなおされる．この場合，scipen を変更することで指数部分に表現しなおされる基準の桁数を変えることが出来る．デフォルトの値は 0 で，値を増やせば基準の桁数が増える.

```
> options(scipen=0);  print(1e5)
[1] 1e+05
> options(scipen=1);  print(1e5)
[1] 100000
```

（例 3）　ベクトルの内容を出力する際、一行あたりの要素の数を少なくする場合は width オプションの値を変更する.

```
> x <- runif(6)
> x
[1] 0.49152403 0.65764434 0.02852227 0.34614703 0.37701605 0.42428670
> options(width=50)
> x
[1] 0.49152403 0.65764434 0.02852227 0.34614703
[5] 0.37701605 0.42428670
```

[*8] この表に挙げた引数は抜粋である．詳しくは help(options) を実行して確認されたい.

第 4 章

ベクトルの基本

R では実数，複素数，文字列，論理数などの基本的データを一つずつ単独で扱う代わりに，同じ型のデータをいくつかまとめたベクトルと呼ばれる形で取り扱っている．よって，今までの例では数値や文字列を一つずつ単独で扱っているかのように説明してきたが，実際には R はこれらの基本的データを，要素の個数が一つだけのベクトルとして扱っていたわけである．例えば，以下の計算は要素がひとつのベクトル同士の足し算を行っていることになる．

```
> 1 + 2
[1] 3
```

$1+2$ だから単に 3 と返せばよいだけのはずだが，よく見ると [1] が前についている．小難しく云えば 1×1 のベクトル (1) と 1×1 のベクトル (2) の足し算を行って，結果として 1×1 のベクトル (3) が返ってきたことになる．[1] は『ベクトルの一つめの要素』という意味である．

4.1　ベクトルの作成

ベクトルを作成する基本的な関数は c() である[*1]．

```
> c(1.0, 2.0, 3.0, 4.0, 5.0)          # 長さ 5 のベクトルを作成する
[1] 1 2 3 4 5
```

ベクトルを変数に代入する場合も <– を用いる．ちなみに，c() という名前の関数があるからといって，オブジェクト名に c を使うことが出来ないというわけではない．よって，以下のようなことも出来る．

```
> ( c <- c(1.0, 2.0, 3.0, 4.0, 5.0) ) # c という名のオブジェクトにベクトルを代入
[1] 1 2 3 4 5
```

ベクトルの要素の個数をベクトルの長さと呼び，ベクトルの長さは関数 length() で調べる．

```
> length(x)                           # 要素が 5 個あるベクトルの長さを調べる
[1] 5
```

関数 c() 以外に規則的なベクトルを生成する関数が多数用意されている．まずは例を示す．

```
> 1:5                                 # 1 から  5 まで 1 づつ増加するベクトルを作る
[1]  1  2  3  4  5
> 3:-3                                # 3 から -3 まで 1 づつ減少するベクトルを作る
[1] 3 2 1 0 -1 -2 -3
```

*1 R では整数と実数を特に区別しない．

```
> rep(1:3, length=8)               # 数列（1 2 3）を長さ 8 になるまで反復生成
[1] 1 2 3 1 2 3 1 2
> seq(1, 10, length=5)             # 1 から 10 までを等分割した長さ 5 のベクトルを作る
[1] 1.00 3.25 5.50 7.75 10.00
> seq(along=c(2,8,5))              # 1:length(c(2,8,5)) と同じ
[1] 1 2 3
```

以下に規則的なベクトルを生成する関数の一覧を挙げる．

関数	説明
a:b	a から b $(a < b)$ までの交差が 1 の数列を生成する．$a > b$ ならば交差が -1 の数列を生成する．
seq(a, b, length $=$ n)	a, b 間を n 等分する等差数列を生成．
seq(a, b, by $=$ c)	a から b まで c づつ増加するベクトルを生成．
sequence(c(3,2))	パターンを持つベクトルを生成する．この場合は (1 2 3 1 2)．
rep(a:b, times $=$ c)	a から b まで 1 づつ増加する数列を c 個生成．c は数列でも可．
rep(a:b, length $=$ c)	a から b まで 1 づつ増加する数列を，長さ c になるまで反復して生成．
unique(x)	ベクトル x 中の反復した値を除いたベクトルを返す．
numeric(n)	0 を n 個並べたベクトルを生成．

4.2　ベクトル同士の計算

同じ長さ同士のベクトルの演算 (算術演算，論理演算，比較演算) は，対応する各要素の演算となる．すなわち，普通の数字と同様に + (加算)，− (減算)， * (積)， / (商)， %/% (整数商)，%% (剰余)，^ (べき乗) が行える．

```
> c(1, 2, 3) + c(4, 5, 6)                    # c(1+4, 2+5, 3+6)
[1] 5 7 9
> c(1.0, 2.0) * c(3.0, 4.0)                  # c(1*3, 2*4)
[1] 3 8
```

ベクトルの長さが違う場合は短い方のベクトルの要素が循環して使用される．このことを利用して，ベクトルの全ての要素に同じ処理を施すことも出来る（以下に例示する）．

```
> c(6.0, 5.0, 4.0, 3.0) - c(2.0, 1.0)       # c(6-2, 5-1, 4-2, 3-1) と同じ
[1] 4 4 2 2
> c(1.0, 2.0, 3.0, 4.0, 5.0) - 1             # 全ての要素から 1 を引く
[1] 0 1 2 3 4
> 1 / 2:5                                    # それぞれの要素の逆数
[1] 0.5000000 0.3333333 0.2500000 0.2000000
```

ベクトル同士を比較演算子 < , == , > などによって比較することも出来る．算術演算の場合と同様，これらの演算子はベクトルとベクトルの対応する要素同士を比較して，後述する論理型ベクトルを結果として返すことになる．

```
> c(1, 20, 300, 4, 50) == c(1, 2, 3, 4, 5)  # 1 == 1, 20 == 2, 30  == 3  ...
[1]  TRUE FALSE FALSE  TRUE FALSE
> c(1, 20, 300, 4, 50) > 60                 # 1 > 60, 20 > 60, 300 >  60 ...
[1] FALSE FALSE  TRUE FALSE FALSE
```

4.3 ベクトル用の関数

18 頁で紹介した数学関数をベクトルに適用すれば，各要素毎に関数が適用されて結果が表示される．

```
> sqrt(1:5)
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

また，数値を要素に持つベクトルの各要素に対して演算を行なう関数には以下のようなものがある．また，基本的な計算の説でも紹介した関数 (sqrt() や cos() など) を適用することで，項別に関数を適用することも出来る．ただし，ベクトルの要素に一つでも NA があると結果も NA になるものもあるので注意しなければならない．

記号	sum()	mean()	var()	median()	cor()	max()	min()
意味	総和	平均	不偏分散	中央値	相関係数	最大値	最小値

記号	prod()	cumsum()	sd()	sort()	rev()	pmax()	pmin()
意味	総積	累積和	標準偏差	昇順整列	要素を逆順	並列最大値	並列最小値

記号	range()	match()	diff()	rank()	order()
意味	範囲	引数のマッチング	前進差分	整列した各要素の順位	整列した各要素の元の位置

まず，一つのベクトルを使った計算例を示す[*2]．

```
> x <- 1:5
> rev(x)                                # x の要素を逆順にする
[1] 5 4 3 2 1
> y <- cumsum(x)                        # y に x の累積和を代入
> mean(y)                               # y の平均を求める
[1] 7
> mean(y, 0.5)                          # y の小さい方と大きい方を合わせて
[1] 6                                   # 100*0.5%=50% を除いた平均 (trimmed mean)
```

次に，複数のベクトルを使った計算例を示す．

```
> y <- c(0, 1, 3, 6, 10);
> sort(y)                                 # y を整列する
[1]  0  1  3  6 10
> order(y)                                # y を整列したときの各要素の元の位置
[1] 1 2 3 4 5
> rank(y)                                 # y を整列したときの各要素の順位
[1] 1 2 3 4 5
> z <- c(100, 20, 4, 2, 1)
> pmax(x, y, z)                           # x, y, z の各要素を比較して一番大きいものを順に返す
[1] 100  20   4   6  10
> pmin(x, y, z)                           # x, y, z の各要素を比較して一番小さいものを順に返す
[1] 0 1 3 2 1
```

[*2] 関数 cumsum() の亜種について，数列の部分和・積、そして部分最大・最小値からなる数列を計算する関数がそれぞれ cumsum()，cumprod(), cummax(), cummin() と用意されている．

4.4 ベクトルを用いた集合演算

ベクトルを集合と見立てて集合演算を行うことも出来る．まず，集合演算用の関数を紹介する．

コマンド	説明
union(x, y)	和集合.
intersect(x, y)	積集合.
setdiff(x, y)	差集合.
setequal(x, y)	集合として等しいか否か.
is.element(x, y)	x 中の各要素は集合 y に含まれるか否か（x%in%y でも可).

次に，計算例を示す.

```
> x <- 1:5;  y <- 1:4
> intersect(x, y)                     # 積集合
[1] 1 2 3 4
> setequal(x, y)                      # 集合として等しいか否か
[1] FALSE
> is.element(x, y)                    # x 中の各要素は集合 y に含まれるか否か
[1]  TRUE  TRUE  TRUE  TRUE FALSE
```

4.5 ベクトル要素へのアクセス

ベクトルの中の数を「要素」と呼び，各要素には左から順に 1, 2, ··· と番号が振られている．以下ではベクトル x について要素にアクセスする方法を一覧表で示している．

コマンド	機能
x[k]	k 番目の要素を取り出す．要素番号として 0 を指定すると，長さ 0 で元のベクトルと同じ型のベクトルが返る.
x[k] <– a	k 番目の要素を a に変更.
x[正整数ベクトル]	いくつかの要素をまとめて取り出す.
x[負整数ベクトル]	対応する要素番号の要素を取り除く.
x[論理値ベクトル]	TRUE の要素に対応した要素を取り出す.
x[条件式]	条件に合致した要素を取り出す.
x[文字型ベクトル]	要素ラベルを指定して要素を取り出す（names 属性が付いている場合).

以下に簡単な例を示す[*3].

```
> x <- c(1, 2, 3, 4, 5)
> x[3]                      # 3 番目の要素を取り出す
[1] 3
> x[5] <- 0                 # 5 番目の要素を 0 に変更する
> x                         # x の値を確認する
[1] 1 2 3 4 0
```

[*3] Mathematica や Maple などの数式処理ソフトでは，負値のベクトルを指定した場合の動作が R と異なるので注意.

次に，多少込み入った例を示す[*4]．

```
> x <- c(1, 2, 3, 4, 5)
> x[2:5]                              # 2 番目～ 5 番目の要素を取り出す
[1] 2 3 4 5
> x[c(-1,-3)]                         # 1 番目と 3 番目以外の要素を取り出す
[1] 2 4 5
> x[30 < x]                           # 30 より大きい要素のみを取り出す
[1] 40 50
> x[10 < x & x < 40]                  # 2 つの条件は & や | で繋ぐ
[1] 20 30
> (1:length(x))[10 < x & x < 40]     # 何番目の要素が条件を満たしているか
[1] 2 3
```

関数 which() を用いることで，ベクトルからある値に最も近い要素の番号を求めることが出来る[*5]．

```
> x <- 1:10
> num <- which( abs(x-5.2) == min(abs(x-5.2)) )   # 5.2 に最も近い値を持つ要素の番号
> x[num]                                          # 該当する要素の値
[1] 5
```

4.6　ベクトル要素の置換・結合・挿入

ベクトルの要素の一部を置き換える場合は関数 replace() を用いることで実現できる．

```
> x <- c(1, 2, 3, 4, 5)                # x <- 1:5 でもよい
> ( y <- replace(x, c(2,4), c(0,0)) )  # x の 2, 4 番目の要素を 0 に置き換える
[1] 1 0 3 0 5
> ( z <- replace(x, c(2,4), NA) )      # 今度は NA で置き換える
[1]  1 NA  3 NA  5
> replace(z, which(is.na(z)), 10)      # z の NA を 10 で置き換え
[1]  1 10  3  4 10  6  7 10  9
```

2 個以上のベクトルを結合する場合は関数 c() を用いる．2 つのベクトルを結合するだけならば関数 append() でも実現出来る．また，関数 append() を用いることで，ベクトルの指定の場所にベクトルを挿入させることも出来る．

```
> x <- c(1, 2, 3);  y <- c(4, 5, 6);  z <- c(7, 8, 9)
> c(x, y, z)
[1] 1 2 3 4 5 6 7 8 9
> ( w <- append(x, z) )    # c(x, z) でもよい
[1] 1 2 3 7 8 9
> append(w, y, after=3)    # w の 3 番目の要素の後に y を挿入
[1] 1 2 3 4 5 6 7 8 9
```

[*4] & や | , == の詳しい解説は 70 頁を参照のこと．ところで，[と] の間のベクトルの長さが元のベクトルの長さよりも小さければ巡回的に評価が繰り返され，ベクトルの長さが長すぎるときはその部分の評価結果には NA が出力される．ただし，正値と負値が混在したベクトルは指定出来ない．

[*5] ベクトルの最大・最小要素の位置を求める関数として which.max(), which.min() が用意されている．

4.7　種々のベクトル

ベクトルの要素には実数以外を用いることも出来る．

4.7.1　複素型ベクトル

複素型ベクトルは複素数を要素とするベクトルのことである[*6]．実部と虚部をベクトルで指定したり，絶対値と偏角を指定することでも複素型ベクトルを作ることが出来る．

```
> c(1+1i, 2+3i)                         # 虚数 1+i を表すときは 1+1i と表記する
[1] 1+1i 2+3i                           # 1+i とすると i は変数と認識される
> complex(re=1:3, im=4:6)               # re ：実部 ，im ：虚部
[1] 1+4i 2+5i 3+6i
> complex(mod=c(1,2), arg=c(0, pi))     # mod：絶対値，arg：偏角
[1]  1+0.000000e+00i -2+2.449213e-16i
```

複素数を要素にもつベクトルを処理する場合は，実数同士の計算時に用いた関数に加えて，以下の関数を用いることが出来る．

記号	Re(z)	Im(z)	Mod(z)	abs(z)	Arg(z)	Conj(z)
意味	実部	虚部	絶対値	絶対値	偏角	共役複素数

4.7.2　論理型ベクトル

論理値を要素とするベクトルを論理型ベクトルと呼ぶ．正式名（TRUE，FALSE）でも略記名（T，F）でも指定できる．論理型ベクトルに対する論理演算子として，& (論理積)，| (論理和)，! (否定)，xor (排他的論理和) があり，これらを用いると要素毎の演算を行なう．2 つのベクトルの長さが異なる場合は短い方の要素が循環的に使われる．

```
>  c(T, F, T) | c(TRUE, FALSE, FALSE)
[1]  TRUE FALSE  TRUE
> !c(T, F, T)
[1] FALSE  TRUE FALSE
> xor( c(TRUE, FALSE, TRUE), c(TRUE, FALSE, FALSE) )
[1] FALSE FALSE  TRUE
```

論理ベクトル全体を調べる関数として以下の様なものがある．

関数	機能
any()	与えられた論理ベクトルの要素に TRUE が一つでもあれば TRUE を返す．
all()	与えられた論理ベクトルの要素がすべて TRUE なら TRUE を返す．

```
> x <- seq(0, 1, 0.2)
> any(x > 0.8)          # x の中に 1 つでも 0.8 以上の要素があるか
[1] TRUE
> all(x < 0.8)          # x の要素は全て 0.8 以下であるか
[1] FALSE
```

[*6] ベクトルの要素中に 1 つでも複素数が見つかれば要素全体が複素数となる．

4.7.3 文字型ベクトル

文字列を要素とするベクトルを文字型ベクトルと呼ぶ.

```
> c("HOKKAIDO", "SENDAI", "TOKYO", "NAGOYA", "OSAKA", "HAKATA")
[1] "HOKKAIDO" "SENDAI"   "TOKYO"   "NAGOYA"   "OSAKA"   "HAKATA"
```

文字列には辞書式順序（ASCII 配列）による大小関係があるので，この規則に従った文字型ベクトル同士の比較演算を行なうことが出来る.

```
> "TOKYO" > "OSAKA"
[1] TRUE
> "HAKATA" > "OSAKA"
[1] FALSE
```

4.7.4 順序つき因子ベクトルと順序無し因子ベクトル

関数 factor() を用いることで，カテゴリーを要素としたベクトルを作成することが出来る．関数 levels() でグループ化されているかを確認することが出来，関数 str() でオブジェクトの要約値を表示することが出来る.

```
> x <- rep(c("L", "M", "H"), 9)
> ( fc <- factor(x) )              # 要素をグループ化
 [1] L M H L M H L M H L M H L M H L M H L M H L M H L M H
Levels: H L M
> levels(fc)                       # グループ化されているかを確認（"L", "M", "H" の順ではない...）
[1] "H" "L" "M"
```

結果を見ると，要素の順序が勝手に入れ替わっていることが分かる．明示的に順序を指定する場合は，引数 levels に順序を指定すれば良い.

```
> ( fc <- factor(x, levels=c("L", "M", "H")) )
 [1] L M H L M H L M H L M H L M H L M H L M H L M H L M H
Levels: L M H
```

関数 factor() では因子間に大小関係は無かったが，関数 ordered() で順序付きの因子ベクトルが作成出来る.

```
> fc <- ordered(x)
> str(fc)                          # オブジェクトの要約値を表示（大小関係があることが分かる）
 Ord.factor w/ 3 levels "H"<"L"<"M": 2 3 1 2 3 1 2 3 1 2 ...
```

関数 gl(カテゴリー数, 繰り返し数, ラベル又は要素数) でパターンを持った因子ベクトルを生成することが出来る.

```
> gl(2, 10, 20)
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
Levels: 1 2
> gl(2, 8, label = c("Control", "Treat"))
[1] Control Control Control Control Control Control Control Control
[9] Treat   Treat   Treat   Treat   Treat   Treat   Treat   Treat
Levels: Control Treat
```

4.8 文字列を操作する

4.8.1 文字列ベクトルの結合

複数の文字列を連結して一つの文字列にするには関数 paste() を用いる．引数がベクトルならばそれぞれの要素ごとに連結することになる[*7].

```
> paste("May I", "help you ?")
[1] "May I help you ?"
> paste(month.abb, 1:12, c("st", "nd", "rd", rep("th",9)) )
[1]  "Jan 1 st"  "Feb 2 nd"  "Mar 3 rd"  "Apr 4 th"  "May 5 th" "Jun 6 th"
[7]  "Jul 7 th"  "Aug 8 th"  "Sep 9 th" "Oct 10 th" "Nov 11 th" "Dec 12 th"
```

2 つの文字列を 1 つに結合する際，間に挟む文字列は引数 sep で指定する (空文字列 ”” を与えれば何も挟まずにつなぐ)．また，ベクトルの各要素を一つにつなげた文字列を作りたいときは，引数 collapse に間に挟む文字 (列) を指定すればよい（sep と同じように 空 ”” で与えてもよい).

```
> paste("/usr", "local", "bin", sep = "/")   # / を間に挟んでつなぐ
[1] "/usr/local/bin"
> ( x <- paste(1:4) )
[1] "1" "2" "3" "4"
> paste(x, collapse="abc")
[1] "1abc2abc3abc4"
> paste(x, collapse="")
[1] "1234"
```

4.8.2 部分文字列の取り出し

部分文字列を取り出す際は，関数 substring() を用いればよい．使い方は以下の通り．

```
> substring("abcdefg", 2, 5)    # substring(文字列 ， 要素の最初の位置，要素の最後の位置)
[1] "bcde"
```

4.8.3 文字列を R の命令として実行

関数 eval(parse(text=”文字列”)) とすればよい．他にも関数 eval.parent(parse(text=”文字列”)) が用意されている．

```
> a <- numeric(5);  x <- 1:5
> for(i in x) eval(parse(text=paste("a[",i,"] <- 10", sep="") ))
> a
[1] 10 10 10 10 10
```

[*7] R に標準で用意されている定数は LETTERS (アルファベットの大文字)，letters (アルファベットの小文字)，month.abb (月の名前の略称)，month.name (月の名前)，pi (円周率) の 5 つがある．

他にも文字列に関する関数が多数用意されている．

記号	charmatch()	chartr()	grep(), match()	gsub(), sub()
意味	文字列の部分的マッチング	文字の置換	パターンマッチ	置換
記号	nchar()	pmatch()	toupper()	tolower()
意味	文字数 (バイト数)	部分マッチング	大文字に置換	小文字に置換
記号	strsplit()	strwrap()	substr()	
意味	正規表現による文字列の分割	文字列 (英文) の分割	部分文字列の切り出し	

4.9 NULL，NA，NaN，Inf の操作

4.9.1 NULL，NA，NaN，Inf なのか否かを調べる

NULL (何も無い)，NA (欠損値)，NaN (非数)，Inf (無限大) は，大抵は演算を施してもそのままの値（ NA や NAN ）が返ってくる．すなわち，原則として NaN にどのような演算を施しても結果は NaN になる．よって，比較演算子 == すら使えないことになる．

```
> x <- c(1.0, NA, 3.0, 4.0)  # NA はどれかを調べても...
> x == NA                    # NA に対する演算は全て NA となる
[1] NA NA NA
```

これら 4 つの値の検査を行う関数がそれぞれ用意されている．例えば，ある値が NA かどうかのテストは関数 is.na() で行なう (比較演算子 == では行なえないことに注意) が，これは NaN を代入しても TRUE が返ってしまう．そこで NaN か否か (NA か否かではない) を判定するために is.nan() という関数が用意されている．

命令	is.null()	is.na()	is.nan()	is.finite()	is.infinite()	complete.cases()
対象	NULL	NA	NaN	有限か否か	無限か否か	欠損か否か

```
> is.na(NA)
[1] TRUE
> is.nan(NA)
[1] FALSE
> x <- c(1, NA, 3, 4)
> is.na(x)
[1] FALSE TRUE FALSE FALSE
```

4.9.2 ベクトル要素にある NA の排除・置換

引数に NA が含まれるとエラーになる関数がある．このような関数に NA が含まれるデータを渡す場合の対処法としては以下のようなものがある．

```
> a <- c(1, 2, NA, 4, 5, NA)
> a[!is.na(a)]                 # a から NA を取り除いた
[1] 1 2 4 5
> a <- ifelse(is.na(a), 0, a)   # NA を 0 に置き換える
> a
[1] 1 2 0 4 5 0
```

NA を置き換えるのではなく，NA を要素に持つベクトルから，NA を読み飛ばして要素を読み取ることを考える．この場合は以下の様にすればよい．

```
> x <- c(1, 2, NA, -4, 5, -6, 7, NA, 9, 10)
> y <- x[!is.na(x)]            # NA 以外の x の要素を y に格納
> y
[1] 1 2 -4 5 -6 7 9 10
```

上を応用すれば，欠損値と負のデータを除去してデータを読み取ることも出来る．

```
> x <- c(1, 2, NA, -4, 5, -6, 7, NA, 9, 10)
> y <- x[(!is.na(x)) & (x > 0)]      # NA と負の値以外の x の要素を
> y                                  # y に格納
[1] 1  2  5  7  9 10
```

4.9.3 NULL の使い方の例

NULL は属性を取り出す関数が属性値がないときの値として返すなど「適当な結果がない」ことを示すために使わる．NULL と欠損値 NA を混同してしまいがちだが，NA はベクトルの要素となり得るが，NULL はベクトルなのでそれ自身でベクトルの要素にはならない点が異なる．

```
> length(NULL)         # モード "NULL" を持つ
[1] 0                  # 長さが 0 のベクトルである
> a <- 1:3
> names(a)             # names 属性が付いていないので NULL が返る
NULL
```

上の手順を逆手にとって，属性値に NULL を代入して属性値を除去することができる．

```
> names(a) <- c("a", "b", "c")
> a
a b c
1 2 3
> names(a) <- NULL
> a
[1] 1 2 3
```

他に，繰り返し文で処理を行うオブジェクトの初期値にも NULL が使われたりする[*8]．

```
> x <- NULL                           # 長さ 0 のベクトル x を用意
> for (i in 1:10) x <- append(x, i*i)  # x に i^2 を順に格納
> x
[1]   1   4   9  16  25  36  49  64  81 100
```

[*8] 他に x <− c() や x <− numeric(0) などが使われる．また，as.character，as.numeric などの型変換関数を使って NULL をある型のベクトルに変換すると，指定された型の長さが 0 のベクトルになる．

第 5 章

行列

5.1　行列の作成

R では以下の手順で行列を作成する[*1].

(1) 行列の要素をベクトル（配列やリスト，行列でも可）で用意する
(2) 関数 matrix(ベクトル, 行数, 列数) でベクトルから行列に変換する

例として，ベクトル (1, 2, 3, 4, 5, 6) を変換して，行列 $\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$ を作成する.

```
> matrix(1:6, nrow=2, ncol=3)          # nrow で行数，ncol で列数を指定する
     [,1] [,2] [,3]                    # matrix(1:6, 2, 3) と略記しても良い
[1,]    1    3    5
[2,]    2    4    6
```

行列を作成する場合，指定された要素は左の列から順に (上から下へ) 埋められる．要素を上の行から順に (左から右へ) 埋める場合は，引数 byrow=T を指定する.

```
> matrix(1:6, nrow=2, ncol=3, byrow=T)  # matrix(1:6, 2, 3, byrow=T) でも可
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

5.2　行列要素へのアクセス

行列の中の数を「要素」と呼び，行は左から順に 1, 2, ··· と，列は上から順に 1, 2, ··· と番号が振られている[*2]. 以下では行列 x について要素にアクセスする方法を一覧表で示している．基本は，[と] の間にカンマで区切って行の位置（番号）と列の位置（番号）を指定すればよい.

```
> ( x <- matrix(1:6, nrow=2, ncol=3) )
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

[*1] R での行列とは行と列を持つ 2 次元配列を指す．数学の世界の行列は各要素が数値となっている行列しか扱わないだろうが，R の行列は 2 次元配列なので論理値や文字列などを要素とする行列を作成することも出来る．よって，配列を生成しているという意味で関数 array() を用いても作成することが出来る.

[*2] x[k] で指定する場合は，要素はまず 1 列目の上から順に 1, 2, ··· ，次に 2 列目の上から順に ··· と番号が振られることになる.

```
> x[1, 2]
[1] 3
> x[-1, c(T,F,T)]                       # 1 行目を非表示，2 行目 1,3 成分を指定
[1] 2 6
```

以下に指定例の一覧を示す.

コマンド	機能	コマンド	機能
x[2,]	2 行目を取り出す.	rowSums()	行の総和
x[,2]	2 列目を取り出す	colSums()	列の総和
x[1, 2]	1 行 2 列目の成分を 取り出す.	rowMeans()	行の平均
x[c(1,2), 2]	1,2 行 2 列目の成分を取り出す.	colMeans()	列の平均
x[c(1,2), c(1,3)]	1, 2 行目と 1, 3 列を取り出す.		
x[,−c(1, 3)]	1, 3 列を除外した行列を取り出す.		
x[, c(T,F,T)]	1, 3 列を取り出す.		

[と] の間にベクトルや行列を指定して行列の一部分を取り出すことも出来る.

```
> x[matrix(c(1,2,2,3), 2, 2)]           # 行列で指定
[1] 3 6                                 # x[cbind(c(1,2), c(3,4))] でも可
> x[1:4]                                # 行列がベクトルに変換されてから
[1] 1 2 3 4                             # 最初の 4 個の要素が取り出される
```

要素を取り出す際，引数に drop=FALSE を指定することで，行列の要素抽出における結果のベクトル化を防ぐことも出来る (結果が複数行で無い場合にベクトル化がなされる) .

```
> x[ ,2, drop=F]                        # 2 列目を行列の形で取り出す
     [,1]
[1,]    3
[2,]    4
```

5.3 行列の結合

ベクトルや行列を結合することで新たな行列を作ることも出来る．行ベクトル単位で結合する場合は関数 rbind() を，列ベクトル単位で結合する場合は関数 cbind() を使う.

```
> ( x <- rbind(c(1,2,3), c(4,5,6)) )    # 行ベクトルを与えて行列生成
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> ( x <- cbind(c(1,2,3),c(4,5,6))  )    # 列ベクトルを与えて行列生成
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

3 つ以上のベクトルや行列を一度に結合することも出来る．

```
> ( x <- rbind(1:3, c(4, 5, 6), 7:9) )  # 3 個の行ベクトルを結合
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> ( y <- cbind(1:3, c(4, 5, 6), 7:9) )  # 3 個の列ベクトルを結合
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

5.4 行列の計算

行列の和，差，積 などの基本的な行列演算は普通の数値と同じコマンドで行うことが出来る[*3]．

```
> a <- matrix(1:4, 2, 2)                # 2 * 2 行列
> b <- matrix(0:3, 2, 2)                # 2 * 2 行列
> a + b - a %*% b                       # 和・差・積
> a * b                                 # 要素ごとの掛け算
> a %o% b                               # 外積（関数 outer(x,y) と同じ）
> a %x% b                               # クロネッカー積
```

（注意） 行列の積は %*% 演算子によって求める点に注意．行列に対して * 演算子を用いると要素毎の積を計算してしまう．

（参考） 関数 outer(x, y, FUN) の引数 FUN に 2 変数関数を指定することで，外積を求める際の関数を指定することが出来る．

```
> x <- seq(-3,3,0.1); y <- seq(-3,3,0.1)
> f <- function(x,y) 1/(2*pi)*exp(-(x^2+y^2)/2)
> z <- outer(x,y,f)   # この後，persp(x,y,z,expand=0.5) などで 3 次元のグラフが描ける
```

演算子 * については，定数部分にベクトルを持ってくることも出来る．また，似たような働きで / を用いることで，要素ごとの逆数を求めることも出来る．

```
> a <- matrix(c(1,2,3,4), 2, 2)         # 2 * 2 行列
> a * (1:2)                             # (1:2) * a も同じ結果
     [,1] [,2]
[1,]    1    3
[2,]    4    8
> 1/a
     [,1]      [,2]
[1,]  1.0 0.3333333
[2,]  0.5 0.2500000
```

[*3] クロネッカー積は関数 kronecker() でも計算できる．

5.5 行列操作方法のカタログ

R には行列操作を行う関数が多数用意されている．以下に一覧表を示す．

コマンド	機能
matrix(0,nrow=2,ncol=3)	2 行 3 列のゼロ行列を作る．matrix(1,nrow=2,ncol=3) と応用することも可．
diag(3)	3×3 の単位行列を作る．diag(1, ncol=2, nrow=2), diag(1, 3), diag(rep(1, 3)), diag(c(1,1)) などでも可．
diag(1:3)	対角成分が (1 2 3) の 3×3 対角行列を作る．
diag(X) <– 1	行列 X の対角成分を全て 1 にする．
diag(x) <– c(1, 2)	2×2 行列 X の対角成分を (1 2) にする．
t(X)	行列 X を転置する．
X[upper.tri(X)] <– 0	上三角成分 (対角成分含まず) を全て 0 にする．対角成分も 0 にする場合は upper.tri(X,diag=TRUE) とする．
X[lower.tri(X)] <– 0	下三角成分 (対角成分含まず) を全て 0 にする．
sum(Xˆ2)	行列成分の二乗．sum(diag(t(X)%*%X)) でも可．
solve(X)	行列 X の逆行列を求める．
ginv(X)	行列 X のムーア・ペンローズ型一般化逆行列を求める（ library(MASS) を実行する必要あり）．
crossprod(X)	行列 X 自身のクロス積を求める（t(X)%*%X)．
crossprod(X,Y)	行列 X と Y のクロス積を求める（t(X)%*%Y)．
eigen(X)	行列 X の固有値と固有ベクトルを求める．
qr(X)	行列 X の QR 分解を行う．
chol(X)	正値対称行列（エルミート行列）X のコレスキー分解を行う．すなわち，X = t(A) %*% A を満たす上三角行列 A を求める．
svd(X)	行列 X の特異値分解を行なう．すなわち，X = U %*% D %*% t を満たす「直交行列 U 」,「X の特異値を対角成分とする対角行列 D 」,「直交行列 V 」を求める．
det(X)	行列 X の行列式を求める．prod(svd(X)$d) でも可．

次節より，一覧表では伝え切れなかった事項を紹介する．

5.5.1 対称行列

三角行列から対称行列を生成することが出来る．

```
> ( x <- matrix(c(1,2,0,3), 2, 2) )
     [,1] [,2]
[1,]    1    0
[2,]    2    3
> y <- x+t(x)                # 三角行列＋三角行列の転置行列
> diag(y) <- diag(y)/2       # 対角成分を元に戻す
> y                          # y に対称行列が出来ている
     [,1] [,2]
[1,]    1    2
[2,]    2    3
```

5.5.2　連立方程式の解

A.x = b なる形の x についての連立方程式の解は関数 solve() を使って解く事が出来る．

```
> a <- matrix(c(0,1,2,3,4,5,6,7,9), 3,3)       #      3y + 6z =  1
> b <- matrix(c(1,0,-2))                       #  x + 4y + 7z =  0
> solve(a,b)                                   # 2x + 5y + 9z = -2
          [,1]
[1,] -2.333333
[2,]  2.333333
[3,] -1.000000
```

係数行列が上三角行列の場合は関数 backsolve() を使う．backsolve() は係数行列の下三角部分を無視するので下三角部分が 0 でない行列を係数行列に指定することも出来るが，この場合に連立一次方程式の係数として使われるのは上三角の部分のみとなる．また，下三角行列の場合は関数 forwardsolve() を用いる．

```
> r <- rbind(c(1,2,3),                         # x + 2y + 3z = 8
+            c(0,1,1),                         #      y +  z = 4
+            c(0,0,2))                         #          2z = 2
> ( y <- backsolve(r, x <- c(8,4,2)) )         #
> r %*% y                                      # 結果は (8   4  2)
> ( y2 <- backsolve(r, x, transpose = TRUE))   # 結果は (8 -12 -5)
> all(t(r) %*% y2 == x)
> all(y  == backsolve(t(r), x, upper = FALSE, transpose = TRUE) )
> all(y2 == backsolve(t(r), x, upper = FALSE, transpose = FALSE))
```

5.5.3　固有値と固有ベクトル

行列の固有値 (実正方行列の固有値) と固有ベクトルは関数 eigen() で求められる．このとき，行列のスペクトル分解がリストの成分として返される．

```
> X <- matrix(c(1,2,3,4), 2, 2)
> z <- eigen(X)                                # X の固有値を求める
> z$values
[1]  5.3722813 -0.3722813
> z$vectors[,1]                                # 固有値  5.3722813 に対する固有ベクトル
[1] -0.5657675 -0.8245648
> z$vectors[,2]                                # 固有値 -0.3722813 に対する固有ベクトル
[1] -0.9093767  0.4159736
```

5.5.4　行列の平方根

関数 svd() を用いると，行列の平方根を求めることが出来る．

```
> A <- array(c(2,1,1,1,2,1,1,1,2), dim=c(3,3))
> U <- svd(A)$u
> V <- svd(A)$v
> D <- diag(sqrt(svd(A)$d))
> B <- U %*% D %*% t(V)                # 行列 A の平方根
> A                                    # 行列 A を表示
     [,1] [,2] [,3]
[1,]    2    1    1
[2,]    1    2    1
[3,]    1    1    2
> B                                    # 行列 B を表示
          [,1]      [,2]      [,3]
[1,] 1.3333333 0.3333333 0.3333333
[2,] 0.3333333 1.3333333 0.3333333
[3,] 0.3333333 0.3333333 1.3333333
> B%*%B                                # 検算（行列 A と一致している）
     [,1] [,2] [,3]
[1,]    2    1    1
[2,]    1    2    1
[3,]    1    1    2
```

5.5.5　正方行列のべき乗

例えば正方行列 A に対する 2 乗操作 A^2 は成分毎に 2 乗した行列を与えるだけで，行列のべき乗 A%*%A は与えてくれない．そこで，A のべき乗 A^n を計算する場合は，A の固有値分解 A = V %*% D %*% t(V) を用いて

$$A^n = \mathrm{V}\ \%{*}\%\ (\mathrm{D\hat{}n})\ \%{*}\%\ \mathrm{t(V)}$$

を計算すればよい．D は対角行列なので D^n と D の行列としての n 乗は一致する．以下では行列の指数乗を計算する関数 matpow() を定義している[*4]．

```
> matpow <- function(x, pow=2) {
+   y <- eigen(x)
+   y$vectors %*% diag( (y$values)^pow ) %*% t(y$vectors)
+ }
> matpow(diag(1:2))          # 行列 ((1, 0) , (0, 2)) の 2 乗
     [,1] [,2]
[1,]    1    0
[2,]    0    4
```

[*4] 関数 matpow() は『RjpWiki』の記事より引用した．

この概念を応用すると行列の指数乗が計算出来る．以下では行列の指数乗を計算する関数 matexp() を定義している．

```
> matexp <- function(x) {
+   y <- eigen(x)
+   y$vectors %*% diag( exp(y$values) ) %*% t(y$vectors)
+ }
> matexp(matrix(0, 2, 2))   # ゼロ行列の指数乗は単位行列
     [,1] [,2]
[1,]    1    0
[2,]    0    1
> matexp(diag(2))           # exp(E) = ((e, 0) , (0, e))
         [,1]     [,2]
[1,] 2.718282 0.000000
[2,] 0.000000 2.718282
```

5.6 その他の行列操作

その他の行列操作の手法を紹介する．面倒ならば読み飛ばしていただいて構わない．

5.6.1 行列の大きさ

行列は dim 属性という次元の属性を持っており，(行数 , 列数) という長さ 2 の整数ベクトルの形をしている．行列に付けられた dim 属性を見る場合は関数 dim() , nrow() , ncol() を用いる．

```
> x <- matrix(1:6, nrow=2, ncol=3)      # 2 * 3 の行列を作る
> dim(x)                                # dim 属性を調べる
[1] 2 3
> nrow(x)                               # 行数 ： dim(x)[1] でも同じ結果が得られる
[1] 2
> ncol(x)                               # 列数 ： dim(x)[2] でも同じ結果が得られる
[1] 3
```

行列の大きさを変えるには，再度 matrix() を使う方法と dim を使って行列の大きさを強制変更する方法とがある．dim を用いる場合の変更結果は，matrix() を使って行列サイズを変更した場合と同様である．

```
> x <- matrix(1:6, nrow=2, ncol=3)      # 2 * 3 の行列を作る
> dim(x) <- c(3, 2)                     # 3 * 2 の行列に強制変換
> matrix(as.vector(x), nrow=2, ncol=6)  # 2 * 6 の行列に強制変換
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    1    3    5
[2,]    2    4    6    2    4    6
> matrix(x, nrow=2, ncol=6)             # matrix() の第 1 引数は
     [,1] [,2] [,3] [,4] [,5] [,6]      # ベクトルに変換されるので
[1,]    1    3    5    1    3    5      # as.vector() は省略可能
[2,]    2    4    6    2    4    6
```

5.6.2 行列要素の補充・置換・抽出

関数 matrix() の引数 nrow , ncol の一方だけを与えると，行列のサイズからもう一方が自動的に決定される．このとき，matrix() に指定した要素の長さが行列のサイズに足りないと，最初の方の要素から循環的に補充される．

```
> x <- matrix(1:6, nrow=3)              # 自動的に ncol = 2 とされる
> y <- matrix(1:6, ncol=2)              # 自動的に nrow = 3 とされる
```

行列作成に使われるベクトルの要素が足りなければ要素が繰り返し使われることは既に述べた．この規則を用いると，以下のような行列を作成することが出来る．

```
> matrix(1, nrow=2, ncol=3)
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
> matrix(1:2, nrow=2, ncol=3)
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
> matrix(1:3, nrow=2, ncol=3, byrow=T)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
```

nrow , ncol に関連して，行列を入れると行番号と列番号で満たされた同じ大きさの行列を返す関数 row() , col() というものもある．

```
> x <- matrix(1:6, nrow=2, ncol=3)       # 2 * 3 の行列を作る
> row(x)                                 # 各行の値が行番号と同じ行列
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
> col(x)                                 # 各列の値が列番号と同じ行列
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
```

要素を置換した結果は，出来るだけ元の行列の形を保存しようとする作用が働く．よって，要素を置換する際に多少の型の不一致があっても自動的に調節される．

```
> ( x <- matrix(1:12, nrow=3, ncol=4) )  # 3 * 4 の行列を作る
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> x[1:2, 3:4] <- c(333, 555, 777, 999)   # 代入するベクトルは 2 * 2 の行列に変換される
```

```
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4  333  777
[2,]    2    5  555  999
[3,]    3    6    9   12
```

ある条件を満たす行列要素の位置（番号）を取り出す場合は関数 which() を，引数 arr.ind=TRUE 付きで使用する．引数で arr.ind=FALSE とすると，行列をベクトル化した上で要素の番号ベクトルを返す．以下では 2 の倍数である要素の行列の位置（番号）を取り出している．

```
> ( x <- matrix(1:6, nrow=2, ncol=3) )
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> ( y <- which(x%%2==0, arr.ind=TRUE) )
     row col
[1,]   2   1
[2,]   2   2
[3,]   2   3
```

ある条件を満たす行列の該当箇所だけアクセスする方法もある．

```
> ( x <- matrix(1:4, 2, 2) )
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> x[x%%2==0]                    # 2 の倍数である x の全要素
[1] 2 4
> x[x%%2==0] <- 0               # 2 の倍数である x の要素を 0 に変更
> x
     [,1] [,2]
[1,]    1    3
[2,]    0    0
```

5.6.3　行（列）に対する演算や操作

関数 apply(X, MARGIN, 関数, ...) を用いると，例えば行列の列 (もしくは行) の和を要素とするベクトルを作成することが出来る．ベクトルや行列，配列の MARGIN に関数を適用し，その結果の配列かリストを返す．ここで MARGIN = 1 ならば行に関して，MARGIN = 2 ならば列に関して，MARGIN = c(1,2) ならば各要素に対して関数を適用する．

```
> ( x <- matrix(1:4, 2, 2) )
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> apply(x, 1, sum)              # 行和 -> 関数 rowSums(x) と同じ
[1] 4 6
```

行（列）に対する演算や操作例の一覧を挙げる．mean や sum を他の関数に変更してもよい．

コマンド	機能
mean(x[i,])	行 i の平均を求める
mean(x[, i])	列 i の平均を求める
mean(x[,])	全要素の平均を求める
apply(x, 1, sum)	各行の和を求める（関数 colSums(x) と同じ）
apply(x, 2, sum)	各列の和を求める（関数 rowSums(x) と同じ）

5.6.4　行列をある行（列）に関してソートする

行列をある行・列に関してソートする場合は関数 order() を用る．また，行列の各列毎の最大要素の位置を求める場合は関数 max.col() を使う．

```
> ( x <- matrix(runif(4), 2, 2) )
          [,1]      [,2]
[1,] 0.9542167 0.7191234
[2,] 0.8059257 0.0016700
> order(x[,2])                    # 2 列目について順序を出力
[1] 2 1
> x[order(x[,2]),]                # 2 列目の数値が昇順になるように並べ替え
          [,1]      [,2]
[1,] 0.8059257 0.0016700
[2,] 0.9542167 0.7191234
> x[,order(x[1,])]                # 1 行目の数値が昇順になるように並べ替え
          [,1]      [,2]
[1,] 0.7191234 0.9542167
[2,] 0.0016700 0.8059257
```

5.6.5　行列イメージを画像で表示

関数 image() を用いることにより，行列のイメージを画像として表示することも出来る．

```
> x <- matrix(rep(0:1,81), 9, 9)
> image(x, col=c(0,9))
```

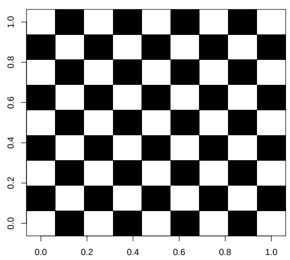


図 5.1　行列のイメージ画像

第 6 章

配列・リスト・データ構造のまとめ

6.1　配列

配列とは行列を多次元に拡張したものである．行列は 2 次元であったが，配列はその次元に応じて 3 , 4 , ... 次元のものを作成することが出来る．配列は以下の手順で作成出来る．

(1) 配列の要素をベクトルなどで用意する
(2) 関数 array(ベクトル, 配列の次元数) でベクトルから配列に変換する

すなわち，配列は関数 array() で作成出来，引数 dim に各次元の要素の個数をベクトルを指定する．また，k 次元の配列は k 個の要素番号でアクセス出来，関数 dim() で配列の次元を調べることが出来る．

```
> ( x <- array(1:8, dim=c(1, 4, 2)) )     # 3 次元配列
, , 1
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
, , 2
     [,1] [,2] [,3] [,4]
[1,]    5    6    7    8
> x[1, 4, ]                               # x[1 次元目の番号，2 次元目の番号，] でアクセス
[1] 4 8
```

配列の要素へのアクセス方法は行列の場合とほぼ同様である．次元ごとに関数を適用する場合も，行列のときの操作方法（57 頁）が使用できる．例えば，3 次元配列 x に対して，全ての配列番号 i について mean(x[i,i,]) を計算するには以下の様にすればよい．

```
> x <- array(1:16, dim=c(2, 4, 2))
> apply(x, c(1,2), mean)
     [,1] [,2] [,3] [,4]
[1,]    5    7    9   11
[2,]    6    8   10   12
```

配列を転置する場合は関数 aperm() を用いればよい．引数 perm (置換) の部分に c(2,1) を入れれば行列の置換と同じになる．例えば以下の様にして使えばよい．

```
> x  <- array(1:24, 2:4)          # 4 次元配列
> xt <- aperm(x, perm=c(2,1,3))   # xt は x を転置した配列
```

6.2　リスト

R にはデータの種類としてベクトルや行列，配列などが用意されている（ 1 や “a” も長さ 1 のベクトル）が，リストはこれら異なる構造のデータを集めて 1 個のオブジェクトにしたものである．異なった型のベクトルを 1 個のリストにまとめてもよいし，リストの要素としてリストを用いても構わない．このリストは関数 list() を用いて作成する．

```
> ( x <- list(1:5, list("It's my list.", c(T, F, T))) )
[[1]]                          # 第 1 成分
[1] 1 2 3 4 5
[[2]]                          # 第 2 成分（リスト）の第 1 成分
[[2]][[1]]
[1] "It's my list."
[[2]][[2]]                     # 第 2 成分（リスト）の第 2 成分
[1]  TRUE FALSE  TRUE
```

リスト x について，操作例の一覧を挙げる．

コマンド	機能
length(x)	リスト x の長さを求める
x <– as.list(NULL)	リスト x を初期化
x[1]	x の第 1 成分を取り出す（中身はリスト）
x[1:2]	第 1，第 2 成分を取り出す（中身はリスト）
x[-1]	第 1 成分以外を取り出す（中身はリスト）
x[c(T,F,T)]	第 1，第 3 成分を取り出す（中身はリスト）
x[[1]]	x の第 1 成分を取り出す（中身はリスト中の要素）
x[2] <– 10	リストの第 2 成分に 10 を代入
x[[2]] <– 10	リストの第 2 成分に 10 を代入
x[3] <– list(NULL)	リストの第 3 成分に NULL を代入する
x[3] <– NULL	リストの第 3 成分を取り除く

上記のように，[[と]] の間に要素の番号を指定することによって，リストの要素を取り出すことが出来る．[と] の間に要素の番号を指定することでもリストの要素を取り出すことは出来るが，この場合はベクトルとしてアクセスを行っていることになる[*1]．よって，要素を取り出したいのではなくてリストの一部分を抽出する場合は，ベクトルの場合と同様，[と] を使えばよい．ただし返り値はリストとなる．

```
> x <- list(1:3, 4:6, 7:9)
> x[1]                                    # 取り出したのはリスト
[[1]]
[1] 1 2 3
```

[*1] [[と]] によって取り出されるものはリストの要素そのものであって，要素を含んだリストではないことに注意．この操作はベクトルの場合の [と] による部分ベクトルの取り出し（範囲指定による抽出）に対応するものではないことに注意．

```
> x[[1]]                             # 取り出したのはリスト中の要素（ベクトル）
[1] 1 2 3
> x[1:2]                             # 第 1，第 2 成分を取り出す
[[1]]                                # （参考）x[[1:2]] は x[[1]][[2]] と同じ（再帰呼出し）
[1] 1 2 3
[[2]]
[1] 4 5 6
```

ベクトルと同様にリストの連結も関数 c() や関数 append() で行なうことが出来る．

```
> c( list(1:5), list(c(T,F,T)) )      # append(list(1:5), list(c(T, F, T))) でも可
[[1]]
[1] 1 2 3 4 5
[[2]]
[1]  TRUE FALSE  TRUE
> rep(list(1:5, "It's my list."), 2)  # 関数 rep() も使用可
```

他にも，リストの要素を端からベクトルとして結合して 1 つのベクトルとしてまとめる関数 unlist() や，逆にベクトルをある基準で区分けしてリストとしてまとめる関数 split() がある．

```
> ( x <- split(1:10, rep(c("odd", "even"), 5)) )
$even
[1]  2  4  6  8 10
$odd
[1] 1 3 5 7 9
> unlist(x)
even1 even2 even3 even4 even5  odd1  odd2  odd3  odd4  odd5
    2     4     6     8    10     1     3     5     7     9
```

6.3　apply() ファミリー

関数 apply() ファミリーには apply(), mapply(), lapply(), sapply(), tapply() が用意されている．一つの関数を複数のオブジェクトに適用して得られた結果をベクトルや行列，リストとして一括で返す．例えば $m \times n$ 行列 X の全ての要素に 1 を足す場合，R では繰り返し文 (for や while) を使わなくても X $<-$ X+1 で手軽かつ高速に実現できるが，この apply() ファミリーに属する関数の機能もこれに似ている．すなわち，C や JAVA ならば複数オブジェクトを個別に与えて繰り返し文で回さなければいけないような場面でも，apply() ファミリーの関数を使えば簡潔な記述で高速に計算できる場面が出てくるということである．

引数	機能
apply(X, MARGIN, 関数, ...)	ベクトルや行列，配列の MARGIN に関数を適用し，その結果の配列かリストを返す．ここで MARGIN = 1 ならば行に関して，MARGIN = 2 ならば列に関して，MARGIN = c(1,2) ならば各要素に対して関数を適用する．
lapply(X, 関数, ...) sapply(X, 関数, ...)	リストに関数を適用し，lapply() は結果のリストを，sapply() は結果の「 names 属性付きのベクトル」か「 names 属性付きの行列」を返す．ベクトルや行列にこれらを適用すると，各成分について関数を適用するのでエラいことになってしまう．

tapply(X, INDEX, 関数, ...)	グループ化された変数について，グループごとに関数を適用する．INDEX は X の要素をグループに分ける因子の組み合わせのリスト (通常は文字列ベクトル) を与え，各グループに関数を適用した結果をベクトルもしくはリストで返す．
mapply(関数 F , x, y, z, ...)	sapply() の多変量版．x, y, z, はベクトルや行列などを複数個指定でき，関数 F(x, y, z, ...) の結果をベクトルのリストで返す．例えば mapply(sum, 1:3, 4:6) ならば [sum(1,4), sum(2,5), sum(3,6)] = [5, 7, 9] が，mapply(sum, 1:3, 4:5) ならば [sum(1,4), sum(2,5), sum(3,4)] = [5, 7, 7] が返ってくる (要素の数が引数ごとで異なる場合は反復して用いられる．ただし warning メッセージが出る.)．
sweep(X, MARGIN, 統計量, FUN="−", ...)	ベクトルや行列，配列の MARGIN で指定した場所から統計量を引く（関数 scale() も同様の機能）．FUN="+" とすれば統計量を足す．

行列の処理でよく使う関数が apply() である．関数 sweep() と組み合わせることで，データの基準化などを行うことも出来る．

```
> ( x <- matrix(1:8, ncol=4) )
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
> apply(x, 2, sum)                          # 各列の最大値を求める
[1]  3  7 11 15
> apply(x, c(1,2), sqrt)                    # 各要素の平方根を求める
         [,1]     [,2]     [,3]     [,4]
[1,] 1.000000 1.732051 2.236068 2.645751
[2,] 1.414214 2.000000 2.449490 2.828427
> sweep(x, 1, apply(x,1,mean))              # 行平均を引く
     [,1] [,2] [,3] [,4]
[1,]   -3   -1    1    3
[2,]   -3   -1    1    3
> sweep(a, 2, apply(a,2,mean))              # 列平均を引く
```

ある関数をリストの要素全てに適用したい場合は関数 lapply()，sapply() を用いる．lapply() を用いた場合は結果は各要素に関数を適用した結果の値のリストで返される．sapply() を用いた場合は結果はベクトルで返されるが，型は要素の中で「一番大きな型」に合わせられる[*2]．

```
> x <- list(a = 1:10, beta = exp(-3:3), logic = c(T,F,F,T))
> lapply(x, mean)                           # リストの三つの成分の平均値からなるリストを計算
$a                                          # 第 1 成分の和
[1] 5.5
$beta                                       # 第 2 成分の和
[1] 4.535125
$logic                                      # 論理値は整数（TRUE=1, FALSE=0）と見なされる
[1] 0.5
```

[*2] 一般的には sapply() の方が計算結果を 2 次使用しやすい．また，sapply() のラッパー関数に replicate() というものがある．詳しくはヘルプを参照されたい．

```
> sapply(x, mean)                          # 型の大小関係で一番大きい方に合わせられる
       a     beta    logic
5.500000 4.535125 0.500000
> lapply(x, quantile, probs = 1:3/4)       # 結果はリスト
> sapply(x, quantile)                      # 結果は属性付きの行列
```

グループ化されたデータの処理でよく使う関数が tapply() である．最初はとっつきにくいが，カテゴリカルデータを扱うは結構役に立つ．

```
> data(warpbreaks)                         # 1 台の織機当たりの反り破損の数
> attach(warpbreaks)                       # 使うデータを固定
> fc <- factor(tension)                    # 要素をグループ化
> levels(fc)                               # グループ化されているかを確認
[1] "L" "M" "H"
> tapply(breaks, fc, mean)                 # breaks の平均をグループごとに計算
       L        M        H
36.38889 26.38889 21.66667
```

トリッキーな使い方かもしれないが，mapply() を使えば規則的なリストを簡単に作成することが出来る．

```
> mapply(rep, 1:3, 3:1)   # rep(1,3), rep(2,2), rep(3,1) を並べる
> mapply(rep, 1:3, 3:1)   # rep(3,1), rep(2,2), rep(1,3) を並べる
```

6.4　データ型の変換とデータ構造の変換

データの型を調べたり変換したりする場合は以下の関数が用意されている．その他，関数 paste() のように必要に応じて引数に与えられたオブジェクトの型変換を自動的に行う関数もある．

	モード	実数	整数	複素数	文字列	論理値
型を検査	mode()	is.numeric()	is.integer()	is.complex()	is.character()	is.logical()
型を変換	storage.mode()	as.numeric()	as.integer()	as.complex()	as.character()	as.logical()

以下に例を示す．

```
> x <- c(1, 0, 1, 0, 1)
> mode(x)
[1] "numeric"
> mode(x) <- "complex"    # storage.mode(x) <- "complex" でも可
> x
[1] 1+0i 0+0i 1+0i 0+0i 1+0i
> mode(x) <- "logical"
> x
[1]  TRUE FALSE  TRUE FALSE  TRUE
```

ベクトルの型の間には以下の大小関係がある．

$$\text{character} > \text{complex} > \text{numeric} > \text{logical} > \text{NULL}$$

ベクトルの要素は全て同じ型なので，異なった型のデータを集めてベクトルを作ろうとすると上記の変換規則によって『最も大きい』型に揃えられる．また，実数と論理数の間は自動的に変換がなされる[*3]．例えば，演算式中に実数値が期待されている時に論理数が表れると以下のように計算される．

$$\text{TRUE} \to 1\ ,\qquad \text{FALSE} \to 0$$

同様に，演算式中に論理数が期待されている時に数が表れると以下のようになる．「 0 でない数 → TRUE 」という点に注意[*4]！

$$0\text{ でない数} \to \text{TRUE}\ ,\qquad 0 \to \text{FALSE}$$

また，データ構造を調べたり変換したりする場合は以下の関数が用意されている．例えば，オブジェクトが配列であるかどうかは is.array() でテスト出来，行列は 2 次元の配列なので is.array() でテストすると結果は TRUE になる．

構造	ベクトル	行列	配列	リスト	データフレーム	順序なし因子	順序つき因子
構造を検査	is.vector()	is.matrix()	is.array()	is.list()	is.data.frame()	is.factor()	is.ordered()
構造を変換	as.vector()	as.matrix()	as.array()	as.list()	as.data.frame()	as.factor()	as.ordered()

ここで例を示す．

```
> x <- c(1, 0, 1, 0, 1, 0)                # x にベクトルを付値
> is.list(x)                              # ベクトルはリストではない
[1] FALSE
> x <- as.list(x)                         # x をリストに変換
> is.list(x)                              # x はリストであり ...
[1] TRUE
> is.vector(x)                            # リストはベクトルでもある
[1] TRUE
```

例示はしないが，as.character() や as.numeric() などで NULL を構造変換すると，指定された型の長さ 0 のベクトルになる．

6.5　names 属性と要素のラベル

ベクトルには names 属性と呼ばれる属性情報を付けることが出来，要素の名前を持つベクトルは名前を使って要素を取り出すことが出来る．

（例 1 ）　mydata の各要素の名前が myname の各文字列に入っている．mydata に name 属性を付け加えるには以下のようにする．ただし names 属性の値はベクトルと同じ長さの文字型ベクトルでなければならない．

```
> mydata <- c(57, 173, 19)                    # 体重，身長，BMI のデータ
> myname <- c("weight", "height", "BMI")      # 名前
> names(mydata) <- myname                     # 名付ける
> mydata                                      # mydata に名前が付いた
weight height    BMI
    57    173     19
```

[*3] 複素数，文字列へも自動的に変換されるが，逆は意識的に行った方が良い．
[*4] 例えば if (2.718) などは TRUE になり，if 文の命令が実行される．

names 属性を付ける事で，ベクトルに名前でアクセスすることが出来る．

```
> mydata["BMI"]
BMI
 19
> mydata["BMI"][[1]]
[1] 19
```

（例 2）　ベクトルと同じようにリストも names 属性を持つことが出来る．names 属性の意味付けもベクトルの場合と同じで，リストの要素の名前として扱われる．

```
> x <- list(57, "173", c(1,9,0,4,5,0,7))        # 体重，身長，BMI のデータ
> names(x) <- c("weight", "height", "BMI")      # 名前を名付ける
> x
$weight
[1] 57
$height
[1] "173"
$BMI
[1] 1 9 0 4 5 0 7
> x$weight                                      # 名前でアクセス（ x[["weight"]] でも可）
[1] 57
```

[[ ]] の動作と$ の動作とでは多少の違いがある．

```
> temp <- "height"                              # temp に文字列 "height" が代入される
> x[[temp]]                                     # [[ ]] の中身の式は評価されるので，
[1] "173"                                       # これは x[["height"]] と認識される
> x$temp                                        # $ の後の式は評価されないので，これは
NULL                                            # x$temp と認識されるので NULL が返る
```

（例 3）　リストの作成時に 名前 = 要素 の形で要素を指定することによって，要素に名前を付けることも出来る．名前は ”” で囲んでも囲まなくてもよい．

```
> list(sequence=1:5, "letters"="abc")
$sequence
[1] 1 2 3 4 5
$letters
[1] "abc"
```

（例 4）　行列に names 属性を与えることによって，ラベルを付けることも出来る．

```
> x <- matrix(1:6, nrow=2, ncol=3)              # 2 * 3 の行列
> rownames(x) <- c("up", "down")                # 行の名前
> colnames(x) <- c("left", "center", "right")   # 列の名前
> x
     left center right
up      1      3     5
down    2      4     6
```

```
> x["up","right"]                                    # 名前で要素にアクセスすることが出来る
[1] 5
```

dimnames 属性を使って行と列にラベルを付けることも出来る．行列に dimnames 属性をつけると、その行列を表示する際にそれが使われる．

```
> a <- matrix(1:6, nrow=2, ncol=3)
> dimnames(a) <- list(c("row1","row2"), c("col1","col2","col3"))
> a
     col1 col2 col3
row1    1    3    5
row2    2    4    6
```

行と列のどちらかのラベルが不要ならば，dimnames 属性の対応する要素を NULL にすればよい．両方のラベルが不要ならば，dimnames 属性自体を取り除けばよい．

```
> rownames(x) <- NULL                                # 行に付けられた名前を全て取り除く
> colnames(x) <- NULL                                # 列に付けられた名前を全て取り除く
> dimnames(x) <- NULL                                # 両方のラベルを取り除く
```

ベクトルを，そのオブジェクト名を用いて行列にする場合は以下のようにする．

```
> x <- 1:3
> y <- 4:6
> matname <- c("x", "y")                             # x, y のオブジェクト名の文字列ベクトル
> sapply(matname, get)                               # x, y を行列に（オブジェクト名＝列ラベル）
     x y
[1,] 1 4
[2,] 2 5
[3,] 3 6
```

（例 5 ）　配列にも行列の場合と同じように dimnames 属性によって names 属性を付けることが出来る．k 次元の配列に対する dimnames 属性は長さ k のリストとなる．ラベルが不要な次元には，要素に NULL を代入すればよい．

```
> a <- array(1:16, dim=c(2, 4, 2))
> dimnames(a) <- list(NULL, c("A", "B", "C", "D"), c("a", "b"))
> a
, , a
     A B C D
[1,] 1 3 5 7
[2,] 2 4 6 8
, , b
      A  B  C  D
[1,]  9 11 13 15
[2,] 10 12 14 16
```

（例 6 ）　行と列のどちらかのラベルが不要なら dimnames 属性の対応する要素を NULL にし，両方のラベルが不要なら dimnames 属性自体を取り除けばよい．

```
> dimnames(a) <- list(c("row1","row2"), NULL)    # 列ラベルを削除
> dimnames(a) <- NULL                            # 全ラベルを削除
```

（参考）　R には本質的属性 (attribute) と呼ばれる付加情報を持つことが出来，オブジェクトは必ず mode と length という 2 つの属性を持っている (names 属性の様に明示的に与える必要は無く，自動的に属性が付けられる)．この mode と length 以外に names 属性や，因子オブジェクト (factor) ならば levels 属性を，行列ならば dim 属性を持つことになる．オブジェクトが持つ mode と length 以外の全ての属性を調べるには関数 attributes() を使い，mode の変更には関数 as. データ型名 () を使う．また，関数 attr() により任意の属性を操作することも出来る．以下の最後の例では，ベクトル x に dim 属性を付けることで，x を 2×2 行列であるかのように振舞わせている．

```
> x <- 1:4
> mode(x)                                        # x の属性を調べる
> attributes(x)                                  # mode と length 以外の全属性を調べる
> attr(x, "dim") <- c(2,2)
```

第 III 部

関数・数値計算篇

第 7 章

関数について

7.1　関数の使用方法

R には特定の機能を果たす命令が多数あり，関数という形で呼び出すことが出来る．例えば関数 date() を実行すると，現在の日付が出力される．

```
> date()
[1] "Mon Jul 21 16:31:52 2003"
```

対数を計算する関数 log(x) は「 x の値の対数を計算する関数」であるが，関数に x の値を知らせる場合には引数という形で関数に与える．

```
> x <- 10                        # x に 10 を代入
> log(x)                         # x を引数に与えて log10(x) を計算させる
[1] 2.302585
```

引数に式や関数を指定すれば，それを評価した値が引数として使われる．また，引数が 2 個以上ある場合はコンマ , で区切って並べればよい．

```
> x <- 10                        # x に 10 を代入
> log(x, base=10)                # 底を指定する引数 base に 10 を指定して対数を計算
[1] 1
```

7.1.1　引数の省略

関数の引数には省略可能なものがある．引数が省略できるか否かは関数のヘルプに載っている．例えば，関数 log() の定義は以下のようになっており，引数 base の値を指定しない場合は，自然対数 e が自動的に指定されることになる．

```
> log(x, base = exp(1))
```

引数名を指定しなくても引数の順番さえあっていればよいので，引数の名前 base を省略して以下のように計算しても良い．

```
> x <- 10
> log(x, 10)                     # log(x, base=10) と同じ
```

名前で引数を指定する場合，順番が違っていてもどの引数が指定されたのか判断出来るので指定位置はどこでもよい．

```
> x <- 10
> log(base=2.718281828, x)
[1] 1
```

関数の中で代入式を書くこともできるが，命令が見にくくなる上に計算効率が悪くなる．よって，「代入」と「関数による計算」の手順は 2 行に分けて書くのが良い．

```
> x <- 10                      ### 良い例
> log(x)                       ###
> log(x <- 10)                 #   悪い例
```

7.1.2　関数の定義を見る

関数がどのような式で定義されているかを見る場合，昔は関数の名前のみ（括弧なし）を入力すれば良かった．例えば関数 order() の定義を見る場合は以下のように入力する[*1]．

```
> order
```

しかし，この方法では定義を見ることが出来ない場合が多い[*2]．この場合，関数 methods() と関数 getS3method() を組み合わせることで関数定義を見ることが出来る．

```
> methods(mean)                                 # 関数の methods 一覧を表示
[1] mean.data.frame mean.Date
[3] mean.default    mean.difftime               # default があることを確認
[5] mean.POSIXct    mean.POSIXlt
> getS3method("mean", "default")                # default 中の method を表示
function (x, trim = 0, na.rm = FALSE, ...)   # mean の関数定義が表示される
{
   ...  （関数定義は省略）
```

7.1.3　複合式

関数を定義する上で必須の概念が複合式である．複合式とはいくつかの式を { と } で括ったものである[*3]．{ と } の式の数は 1 つだけでも構わないし，改行・空白・タブを好きなだけ入れても動作や計算結果には影響しない．また，セミコロンで式の区切りを表すことが出来るが，1 行で 2 つ以上の式を書かないのならば，文の最後にはセミコロンを書かずにそのまま改行しても良い[*4]．複合式の中身は先頭の式から順に評価され，最後の式の値が複合式の値となる．

```
>  {
+    x <- 2 ; y <- 50       # 文の最後にセミコロンをつけてもよいし
+    z <- 10                # 文の最後にセミコロンをつけなくてもよい
+    log(x*y, base=z)       # この式の結果が複合式全体の値として返る
+  }
```

[*1] これは自分で定義した関数，元から R に用意されている関数のどちらにも使える方法である．
[*2] 最悪の場合は，直接ソースコードを読みに行く方法がある・・・．
[*3] これをグループ化という．
[*4] C や JAVA などとはこの点が異なる．

```
[1] 2
```

7.1.4　自作の関数を定義する

自分で関数を定義したい場合の基本的な形式は以下のようなものである．

```
関数名 <- function( 引数 1，・・・ ，引数 n ) {
  <関数本体>
}
```

左下に自作の関数を定義する例を挙げる．ところで，自分で関数を定義する場合は R に既に用意されている関数と同じ名前の関数を定義しないように注意しなければならない．定義してしまった場合，R では警告が表示されないことが多いので気付かないことになる．（右下）．この場合は関数 rm() などでそのオブジェクトを消去する必要がある．

```
> myfunc <- function(x) ifelse(x > 0, 1, -1)
> myfunc(1)
[1] 1
> myfunc(-2)
[1] -1
```

```
> sin <- function(x) ifelse(x > 0, 1, -1)
> sin(1)          # 警告は出ない！
[1] 1
> rm(sin)         # オブジェクトを消去
> sin(1)
[1] 0.841471
```

7.2　演算子

7.2.1　比較演算子・比較演算関数

比較演算子には次のようなものがある．== は単なる = ではないことに注意（ = は「代入」）．以下の演算子で ASCII コード順の辞書式での文字列の比較も出来る．

演算子	==	!=	>=	>	<=	<
意味	等しい	等しくない	$\geq$	$>$	$\leq$	$<$
関数	is.na()	is.nan()	is.null()	is.finite()	is.infinite()	complete.cases()
意味	NA？	NaN？	NULL？	有限？	無限？	欠損なし？

x == y , x != y など，演算子による比較は，x, y が異なったタイプである場合や長さが異なる場合に不具合が生じる場合がある．二つのオブジェクトが『殆んど等しいか』どうかをチェックする場合は，関数 identical() と関数 all.equal() を組み合わせた方法が確実である．『殆んど等しい』の意味はオブジェクトの種類により異なるが，数値ベクトルの場合は誤差の範囲内で一致するかどうかを判断する．

```
> identical(1, 1.)                        # R では 1 と 1.0 はともに倍精度実数で表現
[1] TRUE
> identical(1, as.integer(1))             # as.integer(1) は整数型なので FALSE になる
[1] FALSE
> x <- 1;   y <- 1.0;   z <- 1.00000001;
> if (x == y)                             # 数値誤差を考慮に入れない等号比較
+ identical(all.equal(x, z), TRUE)        # 数値誤差を考慮に入れた等号比較
[1] TRUE
```

7.2.2　論理演算子・論理演算関数

論理演算子には次のようなものがある．

<table>
<tr><td>記号</td><td>!</td><td>&&</td><td>&</td><td>||</td><td>|</td><td>xor(x, y)</td></tr>
<tr><td>意味</td><td>否定</td><td>条件付論理積</td><td>論理積</td><td>条件付論理和</td><td>論理和</td><td>排他的論理和 (→ 関数)</td></tr>
</table>

以下に使い方の例を示す．

```
> x <- 7
> (x > 2) && (x < 4)                        # 「2 より大きい」かつ「4 より小さい」
[1] FALSE
> if ( (x > 2) || (x < 8) ) print("True.")  # 「2 より大きい」または「8 より小さい」ならば...
[1] "True."
```

7.2.3　演算子一覧

ここで演算子一覧を紹介する[*5]．上にあるものほど優先順位が高い（先に処理・計算される）が，括弧 ( ) をつけたものはそこが優先的に計算される．$ と [[ ]] は主にリストの要素にアクセスする場合（例：mylist$element , mylist[[1]]）に用いられ，[ ] は主にベクトルの要素にアクセスする場合（例：x[2]）に用いられる．

<table>
<tr><th>記号</th><th>機能</th></tr>
<tr><td>$</td><td>リストの要素</td></tr>
<tr><td>[, [[</td><td>データの一部分，要素，リストの要素</td></tr>
<tr><td>^ , **</td><td>べき乗</td></tr>
<tr><td>−</td><td>マイナス (単項演算子)</td></tr>
<tr><td>:</td><td>公差 ±1 の等差数列</td></tr>
<tr><td>% 文字列 %（%*% , %in% など）</td><td>特殊二項演算</td></tr>
<tr><td>∗ , /</td><td>積，商</td></tr>
<tr><td>+ , −</td><td>和，差</td></tr>
<tr><td>< , > , <= , >=</td><td>大小比較</td></tr>
<tr><td>== , !=</td><td>等しい，等しくない</td></tr>
<tr><td>!</td><td>否定</td></tr>
<tr><td>&, |, &&, ||</td><td>論理積，論理和，かつ，または</td></tr>
<tr><td>≪−</td><td>永続代入</td></tr>
<tr><td><− , −></td><td>代入</td></tr>
</table>

7.3　条件分岐

7.3.1　条件分岐：if ，else

ある条件で場合分けをして処理を行いたい場合は if 文，else 文を使う．以下の「条件式」には TRUE や FALSE の論理値を一つだけ返す式を入れなければならない．

[*5] 以下の表および内容は『RjpWiki』の記事，isac 社の『S version 4 使用の手引』を参考にして作成した．

```
x <- 2
if (x > 0) {               # if ( 条件式 )
  sum(1:x)                 #  条件式が TRUE  のときに実行される部分
} else {
  x <- -x                  #  条件式が FALSE のときに実行される部分
  sum(1:x)                 #
}
```

（注意 1） if と else を括弧無しで複数行に分けるとエラーとなる（左下：3∼4 行目）．また，} と else の間で改行するとエラーとなる（左下：5∼6 行目）．R は，if (条件式) { 式 } の時点で「一連の式が終わった」と思うので，その次に else が出ると「 if が無いのに else が来た！」と思ってエラーを出す．よって，「一連の式はここからここまで」ということを R に教えるためには，括弧で括って複合式とするか，if の評価が終了して改行するまでに else を書く必要がある（右下）．

```
> a <- rnorm(1)
> if (a >= 0) {
+   if (a == 0) b <- "zero"
+   else b <- a         # エラーが出る！
+ }
+ else b <- -a          # エラーが出る！
```

```
> {
+   if (a >= 0) {
+     if (a == 0) b <- "zero"
+     else b <- a
+   }
+   else b <- -a        # エラーは出ない
+ }
```

以上の理由により，関数定義内で以下の様にやってもエラーは出ない．

```
> myfunc <- function(a) {
+   if (a >= 0) {
+     if (a == 0) b <- "zero"
+     else b <- a              # エラーは出ない
+   }
+   else b <- -a               # エラーは出ない
+   return(c(a,b))
+ }
```

（注意 2） R は型変換のルールにより「0 でない数 → TRUE 」と解釈してしまう．例えば if (3.14) などは TRUE になり，if 文の命令が実行されることになる．よくあるエラーが，if 文の条件 $x <- 2$ をくっ付けて書いてしまったために代入文と勘違いされ，「代入文＝ 0 でない数」となり if の条件式が TRUE になってしまうことである．

```
> x <- -3
> if (x<-2) print(x)     # x<-2 は x に -2 を代入するという意味
[1] 2                    # 0 でないので TRUE になってしまう！
```

演算子や数値，関数などの間には空白 (スペース) を適宜入れる習慣を付けた方が良いという例である．

（参考）　ところで，条件分岐 if や else は以下の様な書き方も出来る．

```
> if (a < 0) {                 # 普通の文中
+   print(0)
+ } else if (a < 1) {
+   print(1)
+ } else print(2)
```

```
> myfunc <- function(a) {    # 関数定義中
+   if (a < 0)  print(0)
+   else if (a < 1) print(1)
+   else print(2)
+ }
```

また，関数 ifelse(条件式, 式 1, 式 2) というものもある．条件式の部分にはオブジェクトを入れることが出来る．

```
> x <- c(-2:3)
> sqrt(x)
[1]      NaN      NaN 0.000000 1.000000 1.414214 1.732051
警告メッセージ：
NaN が生成されました in: sqrt(x)
> sqrt(ifelse(x >= 0, x, NA))
[1]       NA       NA 0.000000 1.000000 1.414214 1.732051
```

7.3.2　条件分岐：switch

条件式の評価結果がケース 1，ケース 2，ケース 3， と多数あり，その結果によって場合分けを行いたい場合には switch 文を使う．例えば文字列で条件を分けた場合の switch 文は左下のような形になる．また，整数で条件を分けた場合の switch 文は右下のようになる．ただし，整数で条件を分けた場合は「一致するものがない時に実行される部分」は指定出来ない点，整数が 1 ... n でなければ NULL が返される点に注意．

```
> a <- 1
+ switch(a,                   # switch(文字列,
+   "1" = print("one"),       #  "1" のときに実行
+   "2" = print("two"),       #  "2" のときに実行
+   print("?")                #  一致するものが
+ )                           #        ない時に実行
[1] "one"
```

```
> a <- 2
+ switch(a,                   # switch(整数,
+   print("one"),             #   1 のときに実行
+   print("two")              #   2 のときに実行
+ )
[1] "two"
```

7.3.3　演算子 &&，||

条件式に 2 つ以上の条件が組み合わさった様な条件判定をさせたい場合は && (かつ) や || (または) を用いる．

```
> a <- 2
> b <- -3
> if ((a > 0) && (b > 0)) {
+   c <- a*b
+ } else c <- -a*b
> c
[1] 6
```

```
> a <- 2
> b <- -3
> if ((a < 0) || (b < 0)) {
+   c <- -a*b
+ } else c <- a*b
> c
[1] 6
```

7.4　繰り返し文

7.4.1　for による繰り返し

ある処理を繰り返し行いたい場合，同じ文を何度も書く代わりに for 文を使う．for (ループ変数 in リスト) でリストの要素の一番目から順に要素をループ変数にスタックして for 文中の式を実行し，リストの要素が無くなった時点で for ループから抜けるという動作をする[*6]．よって繰り返し回数は「リストの要素数」となる．
例えば，ループ範囲に整数値のベクトルを指定すると，x に 1 を 5 回足すことが出来る．

```
> x <- 0
> for (i in 1:5) {                # for（ループ変数 in ベクトルやリスト）
+   x <- x + 1                    # ベクトルやリストの要素が空にならない限り式が繰り返される
+ }
> x
[1] 5
```

（例 1） ループ範囲を表すベクトルやリストの要素は for 文が始まる際に．for 文中でループ範囲を表すベクトルやリストの要素を変更しても，変数に代入される値（オブジェクト）は変化しない．例えば左下のように，ループ変数 x を変更しても i は x の最初の値である 1 から 5 が順に代入される．よって，左下と右下は同じ結果を返すことになる．

```
> x <- 1:5
> for (i in x) x <- append(x, i)
> x
 [1] 1 2 3 4 5 1 2 3 4 5
```

```
> x <- 1:5
> for (i in 1:5) x <- append(x, i)
> x
[1] 1 2 3 4 5 1 2 3 4 5
```

（例 2） for ループのループ範囲には行列や配列，リストを指定することも出来る（実際はベクトルとしてアクセスする)．また，何故か文字ベクトルやグラフィックス用のオブジェクトを指定することも出来る．右下に示す通り，関数 plot() に対応したオブジェクトなら，plot() で順次異なるグラフを書かせたりも可能である．文字や論理値が混在したリストでも構わない（一番上位のモードに揃えられる)．

```
> x <- 0
> y <- matrix(1:10, 2, 5)
> for (i in y) x <- x+1
> x
[1] 10
```

```
> par(mfrow=c(2,2))
> x <- list(1:6, sin, runif(6), dnorm)
> for (i in x) {
+   plot(i, xlim=c(0,2*pi))
+ }
```

7.4.2　while による繰り返し

同じ繰り返しでも，ある条件が成り立っている間中ずっと「繰り返す式」を繰り返す場合は while 文を使う．ただし，条件式が TRUE しかとり得ない場合は，プログラムが永遠に実行され続けて暴走してしまうので注意が必要である．例えば以下を実行すると x の各要素に 1 を 5 回足すことが出来る．

[*6] ごく一般的なプログラム言語の繰り返しアルゴリズムと多少異なる．

```
> x <- 0
> while (x <= 5) {          # while ( 条件式 )
+   x <- x + 1              # 条件式が TRUE である限り式が繰り返される
+ }                         # 最初に条件式が FALSE ならば式は 1 回も実行されない
> x
[1] 6
```

7.4.3　break を用いて繰り返し文から抜ける

break 文によって，for や while 等の「繰り返し文」の途中で繰り返しから抜け*7，next 文によって，for や while 等の「繰り返し文」の途中で，強制的に次の「繰り返し文」に移ることが出来る*8.

```
> x <- 0
> for (i in 1:5) {
+   x <- x + 1
+   if (x == 3) break    # x が 3 になったら
+ }                      # for 文から抜ける
> x
[1] 3
```

```
> x <- 0
> for (i in 1:5) {
+   next                 # x に 1 を足す前に
+   x <- x + 1           # 次の繰り返しが開始
+ }
> x
[1] 0
```

7.4.4　repeat による繰り返し

repeat 文はそのままでは無制限に繰り返しを行なう．処理を止めるには break 文を用いる．

```
> x <- 0
> repeat {
+   if (x <= 5) x <- x+1   # break 文が見つかるまで式が繰り返される
+   else        break      # x > 5 ならば repeat 文から抜ける
+ }
> x
[1] 6
```

7.5　関数の定義

7.5.1　新しい関数の定義

例えば引数の 2 倍を計算する新しい関数 mydouble() を定義する例を挙げる．

```
> mydouble <- function(x) 2*x

> mydouble <- function(x) {
+   return( 2*x )              # 上と同じ関数定義
+ }
> mydouble(2)
[1] 4
```

*7 動作は break 文が評価された時点で最も内側の繰り返しから抜けることになる．

*8 動作は next 文が評価された時点でそれ以降の処理は中止され，「繰り返し文」の先頭に戻って次の繰り返しが開始される．

7.5.2 新しい演算子の定義

新しい演算子を定義することも出来る．以下では R には用意されていないインクリメント演算子 %+=% , %−=% を定義している．以下に示した通り，新しく作成した演算子を定義する場合は演算子を ”” で囲まなければならない．

```
> "%+=%" <- function(x, y) x <<- x + y
> "%-=%" <- function(x, y) x <<- x - y
> x <- 1;
> (x %+=% 2)
[1] 3
> (x %-=% 3)
[1] 0
```

7.5.3 関数の返す値（返り値）

R では手続きも関数もそれぞれ「関数」と呼ぶのだが，本来の関数として用いるのならば返す値を明示的に指定すべきである．返す値を明らかにするには関数 return() を使えばよい．動作は return() が実行された時点で関数の処理が終了し， return() の引数が関数の返す値になる．

返り値が一つの場合： 左下は，引数の 2 倍を計算する前述の関数 mydouble() だが，引数が数値ベクトルでなければ NA を返すように改造してみる．return() が実行されると関数は終了して後の文は実行されないのだが，この働きを関数の途中で抜け出すために用いることも出来る．また，return() を書かなくても最後の文は複合文全体の返り値になるので，右下のように単に値だけを書くことで値を返すことが出来る．

```
> mydouble <- function(x) {
+   if (!is.numeric(x)) return(NA)
+   return(2*x)
+ }
```

```
> mydouble <- function(x) {
+   if (!is.numeric(x)) return(NA)
+   2*x
+ }
```

返り値が複数の場合： 関数 return() に複数の値を指定すると警告が出る．警告を出さないようにするには，関数 return() の引数にリストを与えればよく，複数の返り値を返すことが出来る．このとき，リストの各成分には元の変数名が名前タグとして自動的に付加される[*9]．

```
> mydouble <- function(x) {
+   y <- 2*x
+   return(x, y)
+ }
> mydouble(2)
$x
[1] 2
$y
[1] 4
警告メッセージ：...
```

```
> mydouble <- function(x) {
+   y <- 2*x
+   return( list(x, y) )    # 警告は出ない
+ }
> mydouble(2)
[[1]]
[1] 2
[[2]]
[1] 4
```

[*9] 名前タグを自前で指定するには，明示的に名前付きリストを返り値にすれば良い．

```
> mydouble <- function(x) return( list(input=x, output=2*x) )
```

返り値が関数の場合：　R の関数はオブジェクトを返すので，返り値として関数オブジェクトを返すことも出来る．

```
> mymultiply <- function(x) {
+   return( function (y){ x*y } )
+ }
> result <- mymultiply(2)  # result(y) は関数 2*x
> result(3)
[1] 6
```

7.5.4　画面に返り値を表示しない

関数 invisible() を使うことで，関数の実行結果を自動的に表示しないようにする．

```
> mydouble <- function(x) {
+   return( invisible(2*x) )
+ }
> mydouble(3)                 # 何も表示されない
> y <- mydouble(3)            # 変数に代入して初めて表示できる
> y
[1] 6
```

7.5.5　エラー・警告を表示

関数 stop() によって意図的にエラーを発生させることが出来る．以下では引数が整数でない場合にエラーを発生させている．エラーではなく，関数 warning() によって警告を発生させることも出来る．

```
> mydouble <- function(x) {
+   if (!is.numeric(x)) {
+     stop("not numeric !")
+   }
+   return(2*x)
+ }
> mydouble("a")
以下にエラー mydouble("a") : not numeric !
```

```
> mydouble <- function(x) {
+   if (!is.numeric(x)) {
+     warning("not numeric !")
+     return(NA)
+   }
+   return(2*x)
+ }
> mydouble("a")
[1] NA
警告メッセージ：not numeric ! in: mydouble("a")
```

7.5.6　エラーが起きても作業を続行する

同様の処理を繰り返すようなシミュレーションを行う際，ある段階でエラーが起きても最後まで処理を続けたい場合は関数 try() を用いる．以下の関数 mydouble() は引数が偶数の場合はエラーを出力するが，関数 try() を用いているので，途中でエラーが起きても関数 mydouble() は最後まで実行されている．

```
> mydouble <- function(x) {
+   if (x%%2 == 0) stop("error !")    # 偶数ならばエラーを
+   return(2*x)                       # 奇数ならば 2*x を返す
+ }
> result <- lapply(1:3, function(x){ try(mydouble(x), TRUE) } )
```

```
> result                                          # 途中でエラーが出るが最後まで実行される
```

7.5.7 ローカル変数と永続代入 ≪– について

関数中で値が代入される変数をローカル変数といい，ここでの値は普通の変数（グローバル変数）に影響を与えない．例えば，関数内 <– による代入がなされた変数はその関数の中でのみ有効である．つまり，関数内で <– による代入を行なっても，関数の外部にある同じ名前の変数の値は決して変わることがない．よって関数の内部では外部に及ぼす影響を考えずに変数名を決めてもよいことになる[*10]．

```
> x <- 10                      # グローバル変数 x
> myfunc <- function(y) {      # y は引数（形式パラメータという）
+   x <- y + 999               # x は関数 myfunc() 中のローカル変数
+   return(x)                  # 引数でもローカル変数でもない変数は自由変数と呼ばれる
+ }                            #
> x
[1] 10                         # グローバル変数 x の値は 10
> myfunc(1)
[1] 1000                       # グローバル変数 x には影響を与えない
> x                            # 影響がないかどうかを確認する
[1] 10
```

関数定義の中からグローバル変数に値を代入する場合は永続代入演算子 ≪– を用いる．永続代入演算子 ≪– でグローバル変数に代入をする場合，特に関数中でグローバル変数に代入をする場合は注意が必要である．

```
> x <- 10                      # グローバル変数 x
> myfunc <- function () {
+   x <- 777                   # 関数 myfunc() 中のローカル変数 x
+   x <<- 99                   # グローバル変数 x に 99 を代入する
+   print(x)                   # ローカル変数 x の値を表示
+ }
> myfunc()
[1] 777                        # 関数 myfunc() の返り値（変数 x ）の値
> x                            # 関数 myfunc() 内でグローバル変数 x の値を変えたので，
[1] 99                         # グローバル変数 x の値は 99
```

上の例では，同じ名前だが実体が異なる二つの変数が登場する．関数中では単なる変数 x はローカル変数を指し，これは関数実行後は残らない．一方，グローバル変数 x は関数実行後も残る．すなわち，慣れないうちはグローバル変数とローカル変数を同じ名前にしない方が良いということである．最後に，グローバル変数にアクセスする場合は assign(変数, 値, env=.GlobalEnv) を用いた方が良いという例を挙げる．

[*10] ただし，関数内で代入が行われない『自由変数』は ローカル変数とは違ったふるまいをする．この仕様は「スコープ」と呼ばれる．

```
> myfunc <- function() print(z)   # z は自由変数
> myfunc()                        # z には何も無いのでエラーが出る
以下にエラー print(z) : オブジェクト "z" は存在しません
> z <- 999                        # グローバル変数 z に 999 を代入して
> myfunc()                        # myfunc() を実行したらエラーが出るかと思いきや…
[1] 999                           # ローカル内だけでなくグローバル環境にまで z を探しに行き，999 が表示される
```

R の「スコープ」という概念に興味のある方は，英語版マニュアルの日本語訳「 R 入門」や「RjpWiki」などで学習することをお勧めする．

```
> x <- 10
> myfunc <- function () {
+   x <- c(1, 2)
+   x[2] <<- 0    # この文を
+   print(x)
+ }
> myfunc()
[1] 1 2 3
> x
[1] 10  0  # いつの間にかベクトルに...
```

```
> x <- 10
> myfunc <- function () {
+   x <- 1:3
+   assign(x[2], 2, env=.GlobalEnv)
+   print(x)
+ }
> myfunc()      # ちゃんとエラーが出る
以下にエラー assign(x[2], 2, env = .GlobalEnv) :
        一番目の引数が不正です
```

7.5.8 関数内での関数定義

関数内でも別の関数を定義することも出来る.

```
> myfunc <- function(x) {
+   y <- 2
+   mydouble <- function(a, b) {
+     a*b
+   }
+   mydouble(x, y)
+ }
> myfunc(3)
[1] 6
```

7.5.9 関数についての情報を見る

関数がどのような式で定義されているかを見る場合は，自分で定義した関数を名前のみ（括弧なしで）入力すればよい．また，関数の形式引数を得る場合や形式引数を設定する場合は関数 formals()，関数 alist() を用いればよい．

```
> f <- function(x) 2*x                        # 関数の定義
> f                                           # 関数の定義式を確認
function(x) 2*x                               # 関数 body(f) でも良い
> formals(f)                                  # 関数 f の形式的引数を調べる
$x
> formals(f) <- alist(x=, y=3)                # 形式的引数 x, y を設定（y は既定値 3）
> f                                           # f の定義を確認
function (x, y = 3)
2*x
> formals(f) <- alist(x = , y = 3, ... = )  # 「その他」の引数 ... の設定
> f
function (x, y = 3, ...)
2*x
```

7.5.10　関数終了時の処理

関数の実行終了時は関数の実行前の状態に戻すことが望ましい．例えば，関数中でグラフィックスパラメータを変更する場合，関数の一番最初にパラメータ値を保存し，関数終了時に値を元に戻すことが望ましい（左下）．ただ，左下の関数定義では，もしも関数が途中でエラーが出て異常終了した場合でも，パラメータ値が元に戻らない．この問題は関数 on.exit() を用いて「関数が終了する時の処理」を指定することで解決する（右下）．関数 on.exit() の処理は，関数が正常終了・異常終了に関わらず，関数終了時に必ず実行される．

```
> myfunc <- function(x) {
+   oldpar <- par()          # 保存
+   par(col="red")
+   plot(x)
+   par(oldpar)              # 復帰
+ }
> myfunc(1:10)
```

```
> myfunc <- function(x) {
+   oldpar <- par()          # 保存
+   on.exit(par(oldpar))     # 関数終了時に復帰
+   par(col="red")
+   plot(x)
+ }
> myfunc(1:10)
```

on.exit() で指定した関数終了時の処理を初期化する場合は on.exit() を引数無しで実行すればよい．また，on.exit() の引数 add によって終了処理を追加する（add=T）のか置換する（add=F）のかを指定することが出来る．

```
on.exit(par(oldpar), add=T)     # 終了処理：par(oldpar) を追加する
on.exit(par(oldpar), add=F)     # 終了処理：par(oldpar) のみを実行する
on.exit()                       # 関数終了時の処理を無効にする
```

7.6　引数について

7.6.1　引数のチェックを行う

特定の引数に対して関数の手続きを変更したい場合は，if 文などの条件分岐で場合分けをすればよい[*11]．以下の関数 myprod() は引数 n が 0 の時に 1 を返す．また，引数のチェックを行う場合，関数 ifelse() も有用となる．

```
> myprod <- function(n) {
+   if (n==0) 1
+   else      prod(1:n)
+ }
> myprod(0)
[1] 1
> myprod(5)
[1] 120
```

```
> myprod <- function (n) {
+   ifelse(n==0, 1, prod(1:n))
+ }
> myprod(0)
[1] 1
> myprod(5)
[1] 120
```

7.6.2　引数の省略

組み込み関数には引数の省略を許すものが多くあるが，自分で定義した関数でも引数の省略を許すように作ることが出来る．これを実現する方法は次の 2 つがある．

[*11] 76 頁の「関数の返す値（返り値）」の例も参照されたい．

引数の宣言部で指定する方法

まず，引数の宣言部で，引数に対する指定値が省略された時の値を指定する方法を示す．例えば左下のような関数を定義すると y は省略可能な引数となり，省略した場合の y の値は 1 となる．また，右下のように，省略時の値を指定する部分に任意の好きな式を書くことも出来る上，式の中で他の引数を参照することも可能である．

```
> mymultiply <- function(x, y=1) {
+   return(x*y)
+ }
> mymultiply(2,5)
[1] 10
> mymultiply(2)
[1] 2
```

```
> mymultiply <- function(x, y=x^2) {
+   return(x*y)
+ }
> mymultiply(2,5)
[1] 10
> mymultiply(2)
[1] 8
```

ただし，以下のように引数を互いに参照しあう定義を行った場合，どちらかの引数を与えて関数を呼び出せば問題はないが，引数無しで関数を呼び出すとエラーになる．

```
> myfunc <- function(x=2*y, y=x^2) {
+   return(x*y)
+ }
> myfunc(2)
[1] 8
> myfunc()
以下にエラー myfunc() : 既定引数の再帰的な参照です
```

関数 missing() を使う方法

関数 missing() を使って，ある引数が実際に存在するかどうか，引数が省略されたかどうかを調べる方法もある．関数 missing() は，指定された引数に対する値が省略されていれば TRUE を返す[*12]．また，関数 missing() によって，引数への値が省略されたかどうかを調べることも出来る．右下では，変数 y が省略されたときに関数 missing() と関数 warning() を用いて警告を表示している．

```
> mymultiply <- function(x, y) {
+   if (missing(y)) return(x)
+   else return(x*y)
+ }
> mymultiply(2,5)
[1] 10
> mymultiply(2)
[1] 2
```

```
> mymultiply <- function(x, y=1) {
+   if (missing(y)) warning("y isn't inputted.")
+   return(x*y)
+ }
> mymultiply(3,2)
[1] 6
> mymultiply(2)
[1] 2
警告メッセージ： y isn't inputted. in: mymultiply(2)
```

7.6.3　任意個の引数を受けとる

例えば，関数 c() は引数の個数の制限が無く，2 個でも 10 個でも好きなだけ引数を与えることが出来る．このような任意個の引数を自作する場合，記号 ... で表される特殊な引数を使えばよい．また，関数の引数 ... の後にさらに引数を持つ関数を定義することも出来る．ただし，関数の呼出時にはこのような引数 ... は必ず【 名前 = 値 】の形で与

[*12] ただし，missing(y) は y が関数中で変更されると信頼できなくなる（例：y ← match.arg() の後では必ず FALSE になる）．

えなければならず，引数名の省略も許されないので注意.

```
> f <- function(x, ...) {
+   label <- deparse(
+             substitute(x)
+            )
+   plot(x, xlab=label, ...)
+ }
> x <- 1:10
> f(x, ylab="y")  # 図は省略
```

```
> g <- function(...) {
+   args <- list(...)
+   return(args)
+ }
> f(1:6, "title")
[[1]]
 [1]  1  2  3  4  5  6
[[2]]
[1] "title"
```

```
> h <- function(... , y=1) {
+   2*y
+ }
> h(5)
[1] 2
> h(5,y=2)
[1] 4
> h(y=2)
[1] 4
```

7.6.4　引数に関数を与える

例えば関数 apply() は引数に関数をとることが出来る．これは，R の関数には引数としてオブジェクトを与えており，関数もオブジェクトであることから，以下のように引数に関数オブジェクトを与えることが出来るからである[*13]. これは自分で定義した関数にも当てはまり，右下のように，引数に関数をとることができる関数を自作することも出来る.

```
> apply(matrix(1:4,2),1,function(x) { 2*x } )
     [,1] [,2]
[1,]    2    4
[2,]    6    8
```

```
> f <- function(x, Fun) Fun(x)
> f(3, sqrt)
[1] 1.732051
> g <- function(x, Fun=sqrt) Fun(x)
> g(5)
[1] 2.236068
```

関数 apply() の例のように，関数定義を直接与えることも出来る．また，関数自身がオプション引数をもち、それを特定の値に指定したいときは右下のようにする.

```
> x <- c(1, 4, 9, 16, 25, 36)
> f <- function(x, Fun) Fun(x)
> f(x, F=function(z) {2*z})
[1]  2  8 18 32 50 72
```

```
> mean(x)
[1] 15.16667
> F <- function(x, a=0)   mean(x, a)
> g <- function(x, a, fun) fun(x, a)
> g(x, 0.4, F)
[1] 12.5
```

7.6.5　引数のマッチングと選択

関数の引数は省略形で指定することが出来る．すなわち，綴りの途中までだけで特定の引数であることが認識できれば，引数をある程度省略して記述することが出来る．これを部分的マッチングと呼ぶ．ただし，以下では ”xl” から ”xlab” が認識できずに（”xlim” の可能性もあるので）エラーが出ている.

```
> plot(sin, xlab = "x-title")     # x 軸にタイトルをつける
> plot(sin, xla  = "x-title")     # "xla" でも "xlab" と分かる
> plot(sin, xl   = "x-title")     # "xl" では "xlab" と分からない
Error in if (from == to || length.out < 2) by <- 1 :
        missing value where TRUE/FALSE needed
```

関数の引数に与えられた値が候補のどれに一致するかどうかをチェックする場合は関数 match.arg() を用いる（この

[*13] 以下の内容は『RjpWiki』の記事を参考にして作成した.

とき部分的マッチングが行われる)．ただし，該当する項目がなければエラーが出る．

```
> myfunc <- function(a, b, do = c("sum", "minus", "multi", "div")) {
+   do <- match.arg(do)
+   switch(do, sum   = a+b,       # do（match.arg の返り値）
+              minus = a-b,       # の結果で条件分岐
+              multi = a*b,
+              div   = a/b)
+ }
> myfunc(3, 2, "minus")
[1] 1
> myfunc(3, 2, "mi")              # 引数："mi" でも "minus" だと分かる
[1] 1
> myfunc(3, 2, "m")               # "m" では "minus" だと分からない...
Error in match.arg(do) : ARG should be one of sum, minus, mult, div
```

7.7　再帰呼び出し

ある関数が関数内で自分自身の関数を呼び出すことを再帰呼び出しという．以下に，階乗を求めるプログラムを紹介する．また，関数 Recall() によって，関数の名前に依存せずに再帰呼び出しを行なうことも出来る．左下の例では関数名を変えると本体で自分自身を呼び出しているところも変更しなければならなかったが，右下のように関数 Recall() を使って再帰呼び出しを行なっていれば，その必要はなくなる．

```
> myfactorial <- function(n) {
+   if (n <= 1) return(1)
+   else        return( n * myfactorial(n-1) )
+ }
> myfactorial(5)
[1] 120
```

```
> myfactorial <- function(n) {
+   if (n <= 1) return(1)
+   else        return( n * Recall(n-1) )
+ }
> myfactorial(10)
[1] 3628800
```

7.8　デバッグについて

長めの命令や関数を定義して実行した際に，エラーメッセージが出力されたり期待通りの動作をしないということは少なくない．そのような場合，関数ならばどこに誤りがあるのかを探して見つけて修正しなければならなくなる．この誤りを修正する一連の作業のことをデバッグという．

古典的でかつよくやられている方法に，関数の途中で cat() や print() を使って，バグの原因かもしれない怪しい変数の値を関数の途中で評価途中で表示させる方法がある．この方法は手軽で確実にデバッグできるが，試行錯誤の手間がかかりすぎる欠点がある．例えば関数 myfunc() を定義したときに，変数 x，y，s の途中の値が知りたい場合は以下のようにする．

```
> myfunc <- function(z) {
+   x <- rnorm(1);      y <- rnorm(1)
+   cat("x =", x, "\n"); cat("y =", y, "\n")            # 現在の x と y の値を調べる
+   ifelse( ((x < 0) || (y < 0)), s <- -x*y, s <- x*y )
+   cat("s =", s, "\n")                                 # 現在の s の値を調べる
+   return(s*z)
+ }
```

```
> myfunc(5)
x = 1.305908
y = -0.1477835
s = 0.1929916
[1] 0.9649579
```

便利なことに R にはデバッグを支援するための関数がいくつか用意されている．

関数	機能
browser()	評価の途中で変数を調べる．（デバッグ中に入力できるコマンド → c：関数の実行を継続，n：関数の残りの部分を一行ずつ実行，ls()：生成されたオブジェクトを全て表示，関数中の変数や式：現時点の値を表示，where：現在のスタックを表示，return()：関数評価に戻る，Q：終了）
debug()	デバッグモードに入る．デバッグ中に入力できるコマンドは browser() で使用できるコマンドと同じ．デバッグモードを抜ける場合は関数 undebug() を実行する．
trace()	関数の呼び出しを追跡する．追跡モードを止める場合は関数 untrace() を用いる．詳しくはヘルプを参照．
traceback()	呼び出されたスタックを表示する．詳しくはヘルプを参照．
debugger()	R 用デバッガを起動する．詳しくはヘルプを参照．
try()	関数の途中でエラーが起きても，関数の残りを実行する．
stopifnot()	引数に与えた条件が満たされないときにプログラムをストップする．
system.time()	プログラムの実行時間を計る．
Sys.sleep()	プログラムの実行を指定秒数中断する．

以下に関数 browser() の使用例を挙げる．

```
> myfunc <- function(z) {
+   x <- rnorm(1);   y <- rnorm(1)
+   ifelse( ((x < 0) || (y < 0)), s <- -x*y, s <- x*y )
+   browser()
+   return(s*z)
+ }
> myfunc(5)
Called from: myfunc(5)
Browse[1]> ls()          # 生成されたオブジェクトを全て表示
[1] "s" "x" "y" "z"
Browse[1]> x             # x の値を表示
[1] -0.3039924
Browse[1]> c             # 関数の実行を継続
[1] 0.6327254
```

7.9　落穂ひろい

7.9.1　連番の変数を作成する

他のプログラム言語（例：C 言語）に用意されている配列は x[1]，x[2]，··· という形をしている．R でも関数 assign() と関数 paste() を組み合わせることで，他のプログラム言語の配列と似たような変数名を付ける事が出来る．

```
> for (i in 1:5) assign(paste("A", i, sep=""), i)     # 変数 Ai に i を代入
> A5
[1] 5
```

7.9.2　数値ベクトルの対話的入力：readline()

関数 readline() を用いると，キー入力があるまでプログラムの実行を停止することが出来る．これにより対話的な入力を行うことが出来る[*14]．キー入力があるまでプログラムの実行を停止するのではなく，秒数を指定してプログラムを停止する場合は関数 Sys.sleep(time) を用いればよい．

```
> mytest <- function() {
+   ANS <- readline("1+2? ")
+   if (ANS == "3") cat("Correct!\n")
+   else            cat("Wrong...\n")
+ }
> mytest()
1+2? 3
Correct!
```

```
> mymessage <- function() {
+   Sys.sleep(3)
+   cat("Hello.\n")
+ }
> mymessage()
Hello.
```

readline() を用いることで数値ベクトルを読み込ませるようにすることも出来る．手順は，まず対話的に文字列を入力（任意の分離記号が指定可）させ，次に関数 strsplit() で文字列リストに変換してから関数 unlist() で文字列ベクトル化し，最後に関数 as.numeric() で数値化する．ここで ”,” で区切ったベクトルを入力することが出来る関数 myfunc() を左下に定義してみる．また，オブジェクト名を入力して読み込ませる場合は右下のようにする．

```
> myfunc <- function () {
+   Str <- readline("Enter : ")
+   as.numeric(unlist(strsplit(Str, ",")))
+ }
> x <- myfunc()
Enter : 1,2,3,4,5
> x
[1] 1 2 3 4 5
```

```
> x <- 1:5
> y <- get(readline())
x
> y
 [1]  1  2  3  4  5
```

7.9.3　メニューによる選択：menu()

関数 menu() でメニュー選択を実現できる．引数 choices で選択肢を示す文字列ベクトル, 引数 title でメニューのタイトルとして出力される文字列を指定する．0, 1, 2, ... を選択することが出来，0 を入力したらメニュー選択から抜けることが出来る．以下では選択肢番号により switch 文で実行内容を変えている．

```
> switch(menu(c("1+2", "3*4"), title="0: exit") + 1, cat("Nothing done\n"), 3, 12)
0: exit
1: 1+2
2: 3*4
選択：1
[1] 3
```

[*14] 以下の関数は『RjpWiki』の記事より引用した．

第 8 章

数値計算

R に用意されている「数値計算を行う道具」を紹介する．

8.0　数値演算誤差

R の内部では 10 進数を 2 進数に直して計算しているため，小数計算の種類によっては近似値（循環小数）になり，計算結果にゴミ（数値誤差）が生じる場合がある．このゴミを見えなくする場合は関数 round() を用いればよい．

```
> 0.4-0.2-0.2                      # 普通の結果
[1] 0
> 0.4-0.3-0.1                      # 結果にゴミが残る
[1] 2.775558e-17
> round( 0.4-0.3-0.1 )             # 関数 round() で丸める
[1] 0
```

8.1　丸めについて

R には実数を丸める以下の関数が用意されている．

関数	説明
round(x, digits = 0)	IEEE 式で丸めを行なう（デフォルトは digits = 0 で小数以下を丸める）．
ceiling(x)	x 未満でない最小の整数を返す．
floor(x)	x 以上でない最大の整数を返す．
signif(x, digits = 6)	digits で指定された有効桁数 (デフォルトは 6 桁) に丸める．
trunc(x)	x を「 0 へ向かって」整数化する．

関数 round() の丸め方は四捨五入とは微妙に異なる．IEEE では任意の桁位置における丸め処理法を次のように定めている[*1]．

(1) 一番近い丸め結果候補が 1 つだけなら，その数に丸める．
(2) 一番近い丸め結果候補が 2 つある場合は，末尾が偶数のものに丸める（五入ばかりでなく五捨もあり得る！）．
(3) 丸め処理は 1 段階で行なわなければならない．

この規則は丸めによる誤差が最小になる利点がある．ただし，規則 (2) が適用される場合は通常の四捨五入とは異なる

*1 RjpWiki の記事，『工学のためのデータサイエンス入門』 間瀬 茂 他 著 (数理工学社) から引用した．

結果になる場合がある．また，規則 (3) は，例えば 2.45 は直に 2 と丸めるべきであって，まず 2.5 としてから，次に 3 としてはならないことを主張している．

命令	round(2.4)	round(2.5)	round(3.5)	round(2.51)	round(3.51)	round(3.46)
結果	2	2	4	3	4	3
規則	(1)：四捨	(2)：五捨！	(2)：五入	(1)：五入	(1)：五入	(1) と (3)：四捨

8.2　ガンマ関数

以下のガンマ関数が用意されている．

関数	説明
beta(a, b)	ベータ関数 B(a,b) = (gamma(a)×gamma(b)) / gamma(a+b)．引数 a,b は零および負整数を除く数値（のベクトル）．
lbeta(a, b)	ベータ関数の自然対数．直接計算しており，log(beta(a,b)) として計算しているわけではない．
gamma(x)	ガンマ関数．引数 x は零および負整数を除く数値（のベクトル）．正の整数 n に対する階乗 n! は gamma(n+1) として計算する．
lgamma(x)	ガンマ関数の自然対数．直接計算しており，log(gamma(x)) として計算しているわけではない．
digamma(x)	lgamma の一階微分．
trigamma(x)	lgamma の二階微分．
tetragamma(x)	lgamma の三階微分．
pentagamma(x)	lgamma の四階微分．
choose(n, k)	二項係数 (n 個から k 個を選ぶ場合の数)．gamma(n+1) / (gamma(k+1)×gamma(n-k+1)) として計算しているので n, k が整数でなくても値は得られるが，結果は整数化される．
lchoose(n, k)	二項係数 choose(n,k) の自然対数．直接計算しており，log(choose(n,k)) として計算しているわけではない．

ガンマ関数：$\Gamma(s) = \int_0^\infty x^{s-1}e^{-x}\,dx$ の性質は以下の通り．

①：$\Gamma(1) = 1$
②：$\Gamma(\frac{1}{2}) = \sqrt{\pi}$
③：$\Gamma(s) = (s-1)\Gamma(s-1) \quad (s > 1)$
④：$\Gamma(n) = (n-1)! \qquad (n:\text{整数})$

階乗の計算について，昔は $n!$ を計算する場合は gamma(n+1) としたものだが[*2]，最近になって関数 factorial() というものが出てきたので，これを使って階乗計算をすることが出来るようになった．例えば 3! を計算する場合は以下のようにする．

```
> factorial(3)
[1] 6
```

[*2] prod() という関数を使っても計算できるが，0! = 1 とは計算してくれない．
また，lgamma(n+1) と定義される関数 lfactorial(n) も用意された．

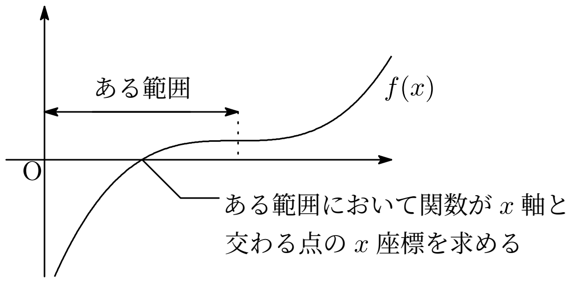
8.3 ベッセル関数

引数 x は非負値，次数 nu は実数（負の値の場合を含む）とする．

関数	説明
besselI(x, nu, expon.scaled=F)	変形された第 1 種のベッセル関数.
besselK(x, nu, expon.scaled=F)	変形された第 2 種のベッセル関数.
besselJ(x, nu)	第 1 種のベッセル関数.
besselY(x, nu)	第 2 種のベッセル関数.
gammaCody(x)	ガンマ関数．gamma() より計算は少し速いが gamma() より精度は落ちる．

8.4 ニュートン法

x の関数 $f(x)$ について，$f(x) = 0$ を満たす解を求める方法をニュートン法という．



R では関数 uniroot() でニュートン法が行える．以下では $0 \leq x \leq 2$ の範囲において，$f(x) = x^3 - 2$ が 0 となる x の値を求めている．

```
> f <- function (x) x^3 - 2                # (1) f(x) を定義
> uniroot(f, c(0, 2))                      # (2) 範囲を c(0, 2) で指定
$root                                      # $root に解が入っている
[1] 1.259934
(以下略)
```

解は $x = 0.6931457$ となっている．

8.5 多項式の解

関数 polyroot() で多項式の解（根）を求めることが出来る．例えば，$p(x) = 5 + 4x + x^2$ の根を求める場合は c(5, 4, 1) を与える．

```
> polyroot(c(5, 4, 1))                     # x^2 + 4x + 5 = 0 の解
[1] -2+1i -2-1i
> polyroot(c(1, 2, 1))                     # x^2 + 2x + 1 = 0 の解
[1] -1-1.665335e-16i -1+1.665335e-16i      # 重解の場合は同じ値が複数返される
```

結果に小さな数値誤差が混じっているが，これを見えなくするには関数 round() を用いればよい．

```
> round( polyroot(c(1, 2, 1)), digits=3 )  # x^2 + 2x + 1 = 0 の解
[1] -1+0i -1+0i
```

8.6 関数の微分

数値計算では，数値微分は誤差が大きくなりやすいので細心の注意を払う必要があるのだが，R では関数を微分することが出来るので，あまり誤差のことを心配する必要はない．

```
> f <- expression( a*x^4 )                 # (1) 関数 f を定義（関数 expression() を用いること）
> D(f, "x")                                # (2) D(f, 微分する変数）で微分する
a * (4 * x^3)
```

高階微分する場合は以下のような関数 DD を定義すると便利である．

```
> DD <- function(expr, name, order = 1) {
+   if(order < 1) stop("'order' must be >= 1")
+   if(order == 1) D(expr, name)
+   else DD(D(expr, name), name, order - 1)
+ }
> DD(f, "x", 3)                            # D(数式, 微分する変数, 微分する回数)
a * (4 * (3 * (2 * x)))
```

結果の数式を使ってさらに計算を行う場合は関数 deriv() を用いればよい．

```
> g <- deriv(~ exp(-x^2), "x", func=T)     # (1) 関数 f = exp(-x^2) を微分，結果を関数 g に代入
> g(2)                                     # (2) g(2) を実行すると・・・

[1] 0.01831564                             #  f(2)  の計算結果
attr(,"gradient")
               x
[1,] -0.07326256                           #  f'(2) の計算結果
```

関数 deriv() を用いた計算例の一覧を示す[*3]．

コマンド	機能
f <– deriv(˜ xˆ2, ”x”, func=T)	x^2 を x で微分したものを関数 $f(x)$ に代入．
g <– deriv(˜ xˆ2∗y, c(”x”,”y”), func = TRUE)	x^2y を「x で微分したもの」「y で微分したもの」それぞれの関数を $g(x, y)$ に代入．
h <– deriv(˜ xˆ2∗y∗z, c(”x”,”y”), function(x, y, z=4))	x^2yz を「x で微分したもの」「y で微分したもの」それぞれの関数を $h(x, y, z = 4)$ に代入．

[*3] deriv() の引数に hessian=T を与えることで hessian を計算することも出来る．さらに，deriv(... , hessian=T) と同じ働きをする関数 deriv3() が用意されている．

8.7　数値積分

数値積分は関数 integrate() で実行出来る．

```
f <- function(x) x^2                        # (1) 積分する関数を定義
integrate(f, 0, 1)                          # (2) integrate(関数，積分範囲の下限，積分範囲の上限)
0.3333333 with absolute error < 3.7e-15     # 結果は 0.3333333
```

R に元々用意されている関数を積分する場合は，新たに関数を定義する必要はない．

```
> integrate(sin, 0, pi)
2 with absolute error < 2.2e-14
```

さらに，無限大（Inf）を範囲に取ること（広義積分）も可能である．以下では正規分布の密度関数を $-$Inf($-\infty$) $\sim$ 1.96 の範囲で積分している．

```
> integrate(dnorm, -Inf, 1.96)         # dnorm:正規分布の密度関数
0.9750021 with absolute error < 1.3e-06
```

パッケージ adapt 中の関数 adapt(次元, lower=積分の下限, upper=積分の上限, functn=関数名) を用いれば多次元の数値積分を行うことが出来る．adapt() の引数 value に積分値が，relerr に相対誤差が表示される．また，関数の引数 eps で相対誤差の最小値を指定することも出来る．

```
> library(adapt)
> mynorm <- function(z) {
+   (1/sqrt(2*pi))^length(z) * exp(-0.5 * sum(z * z))
+ }
> adapt(2, lo = c(-4,-4), up = c(4,4), functn = mynorm)
    value       relerr       minpts       lenwrk        ifail
 0.993709 0.0004751543          263           73            0
```

8.8　関数の最大化・最尤推定法

optim() はデフォルトでは最小化を行なうのだが，最大化をするには制御リスト control の fnscale に負の値を与えればよい (例 ： fnscale=$-$1)．本来の目的関数の代わりに，値にマイナスを掛けた目的関数を用意する方法もある．以下では，ベルヌーイ試行 10 回を行って成功が 3 回であった場合のパラメータ（ここでは確率）p を推定している．

```
> LL < - function(p) choose(10,3)*p^(3)*(1-p)^(7)  # 同時密度分布
> optim(c(0, 1), LL, control=list(fnscale=-1))      # 初期値 (0, 1)
$par
[1] 0.30 0.95
$value            # 関数を最小にするパラメータ値
[1] 0.2668279     # 真の値は 0.3
......................
```

8.9　関数の最小化

関数の最小化を行なう場合は関数 nlm() を用いる．この関数は非線形回帰を行う際によく用いられる[*4]．以下では，ベルヌーイ試行 10 回を行って成功が 3 回であった場合のパラメータ（ここでは確率）p を推定している．

```
> LL <- function(p) -choose(10,3)*p^(3)*(1-p)^(7)  # 同時密度分布
> nlm(LL, 0.5, fscale=-1)                          # 初期値 0.5
$minimum
[1] -0.26683
$estimate         # 関数を最小にするパラメータ値
[1] 0.3           # 真の値は 0.3
.....................
```

8.10　数理計画法

パッケージ lpSolve の中の関数 lp() で数理計画を行うことが出来る．以下では条件（f.con）$x_1 + 2x_2 + 3x_3 \leq 9$, $3x_1 + 2x_2 + 2x_3 \leq 15$ において，式（f.obj）$x_1 + 9x_2 + x_3$ を最大化する例を挙げている．

```
> library(lpSolve)
> f.obj <- c(1, 9, 3)
> f.con <- matrix (c(1, 2, 3, 3, 2, 2), nrow=2, byrow=TRUE)
> f.dir <- c("<=", "<=")
> f.rhs <- c(9, 15)
> lp ("max", f.obj, f.con, f.dir, f.rhs)
Success: the objective function is 40.5
> lp ("max", f.obj, f.con, f.dir, f.rhs)$solution
[1] 0.0 4.5 0.0
```

「変数が整数しかとらない」という条件を付ける場合は以下のようにする．

```
> lp ("max", f.obj, f.con, f.dir, f.rhs, int.vec=1:3)
Success: the objective function is 37
> lp ("max", f.obj, f.con, f.dir, f.rhs, int.vec=1:3)$solution
[1] 1 4 0
```

8.11　高速フーリエ変換

関数 fft() で高速フーリエ変換が行える．また，関数 convolve(x,y) で 2 つの数列の畳み込みが行える[*5]．

```
> x <- 1:4
> fft(x)
[1] 10+0i -2+2i -2+0i -2-2i
> fft(fft(x), inverse = TRUE)/length(x)
[1] 1+0i 2+0i 3+0i 4+0i
```

[*4] 非線形回帰の詳しい解説は『工学のためのデータサイエンス入門』(間瀬 茂 他 著，数理工学社) を参照のこと．

[*5] 関数 mvfft() も参照されたい．

第 IV 部

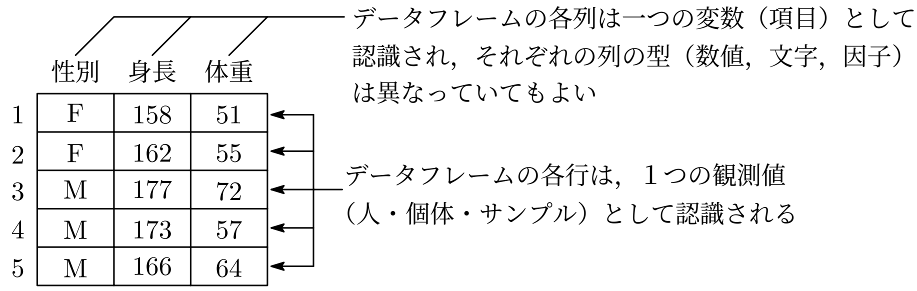
データフレーム篇

第 9 章

ファイルからのデータ入力

9.1　データフレームとは

データフレームとは data.frame クラスを持つリストのことであり，数値ベクトルや文字ベクトル，因子ベクトル（文字型ベクトル）などの異なる型のデータをまとめて 1 つの変数として持っている．外見は行列と同じ 2 次元配列であるが，データフレームの各行・列はラベルを必ず持ち，ラベルによる操作が可能である点が普通の行列と異なる[1]．しかも各列の要素の型はバラバラでも構わないので，ベクトルやリストで持っているデータをデータフレームに変換することで統計解析がやりやすくなる．



9.1.1　データフレーム事始

データフレームを作成する方法は以下のような方法がある．

- ベクトル（や行列，リストなど）からデータフレームを作成する
- ファイルにあるデータを読み込んでデータフレームを作成する

まずは前者の方法を紹介する．「性別」「身長」「体重」データをベクトルで用意しておき，それらを関数 data.frame() で 1 つのデータフレームに変換する[2]．

```
> data.frame(列名 1 = ベクトル 1, 列名 2 = ベクトル 2, ... )
```

実際に関数 data.frame() でデータフレームを作成してみる．

[1] 数値ベクトルと因子はそのままの状態で含まれ，非数値ベクトルは因子に強制変換される．データフレームに変数として現れるベクトル構造は全て同じ「長さ」を，行列構造は同じ「行サイズ」を持たなければならない．

[2] 行列や配列からデータフレームを作成することも出来る．例えば行列 x をデータフレームに変換する場合は data.frame(x) とすればよい．ラベル名を指定しない場合は自動でラベル名が振り分けられる．

```
> sex <- c("F","F","M","M","M")
> height <- c(158,162,177,173,166)
> weight <- c(51,55,72,57,64)
> ( x <- data.frame(SEX=sex, HEIGHT=height, WEIGHT=weight) )
```

```
  SEX HEIGHT WEIGHT
1   F    158     51
2   F    162     55
3   M    177     72
4   M    173     57
5   M    166     64
```

データフレーム x を生成すると，「データフレーム x の『性別』データ」や「データフレーム x の『体重』データ」としてデータを取り出すことが出来る．取り出す方法は「データフレーム名$ 列名」などとすればよい．

```
> x$HEIGHT                    # データフレーム x の『身長』データ
[1] 158 162 177 173 166
> mean( x$WEIGHT )            # データフレーム x の『体重』データの平均
[1] 59.8
> x[,2]                       # データフレーム x の 2 列目を表示する
[1] 158 162 177 173 166
```

9.1.2　データの列の特徴を見る

データフレームという形にすることで，要素それぞれに『性別』や『身長』などの属性を付けることに成功した．その恩恵の一つとして，関数 summary() を使うことでデータフレームの列ごとの特徴を見ることが出来る．すると上から順に「最小値，第 1 四分位点，中央値，平均，第 3 四分位点，最大値」が表示される．

```
> summary(x)
 SEX       HEIGHT          WEIGHT
 F:2   Min.   :158.0   Min.   :51.0
 M:3   1st Qu.:162.0   1st Qu.:55.0
       Median :166.0   Median :57.0
       Mean   :167.2   Mean   :59.8
       3rd Qu.:173.0   3rd Qu.:64.0
       Max.   :177.0   Max.   :72.0
```

9.2　ファイルからデータを読み込む

ある 5 人の健康診断データを紹介する．内容は性別（sex)，身長（height)，体重（weight）のデータである．

sex	height	weight
F	158	51
F	162	55
M	177	72
M	173	57
M	166	64

9.2.0　作業ディレクトリの変更

データを扱う際，まず，データがあるディレクトリ（フォルダ）に作業ディレクトリを変更する必要がある．作業ディレクトリの変更方法は 22 頁を参照されたい．

9.2.1　データフレームの作成（テキストファイルから）

以下に出てくるデータは全て作業ディレクトリにあるものとする．もし，データが作業ディレクトリに無い場合は，ファイルを指定する場所にファイルのパスを指定すればよい（例：x <– read.table(”C:/data.txt”) ）.

(1) data01.txt のようなデータは，関数 read.table() で読む．データファイルに列名が無いので，R が勝手に列名を決めている．

```
> ( x <- read.table("data01.txt") )
  V1  V2 V3
1  F 158 51
2  F 162 55
...........
```

data01.txt

```
  F   158   51
  F   162   55
  M   177   72
  M   173   57
  M   166   64
```

(2) data02.txt のように，1 行目に列名が入っているデータは，関数 read.table() の引数 header（列名があるか否か）に T を指定する．

```
> ( x <- read.table("data02.txt", header=T) )
  sex height weight
1   F    158     51
2   F    162     55
...................
```

data02.txt

```
sex height weight
  F   158     51
  F   162     55
  M   177     72
  M   173     57
  M   166     64
```

(3) data03.txt のように，1 行目にコメント，2 行目に列名が入っているデータは，引数 header に T を，skip（何行読み飛ばすか）に 1 を指定する．

```
> ( x <- read.table("data03.txt", header=T, skip=1) )
  sex height weight
1   F    158     51
2   F    162     55
3   M    177     72
...................
```

data03.txt

```
### data03.txt
sex height weight
  F   158     51
  F   162     55
  M   177     72
  M   173     57
  M   166     64
```

(4) data04.txt のように，データ間がコンマ（ , ）で区切られている場合は，引数 sep（データの区切り文字）に ”,” を指定する．また，タブで区切られている場合は ”¥t” を指定すればよい．

```
> ( x <- read.table("data04.txt", header=T, sep=",") )
  sex height weight
1   F    158     51
2   F    162     55
...................
```

data04.txt

```
sex,height,weight
F,158,51
F,162,55
M,177,72
M,173,57
M,166,64
```

関数 read.table() の引数を以下に示す．例えば，row.names と col.names にそれぞれ行名と列名を示す文字型ベクトルを指定することができる[*3]．

---

*3　関数 rownames(データフレーム名) と colnames(データフレーム名) で，それぞれ行ラベル，列ラベルを取り出すことも出来る．

引数	機能
sep = ””	データとデータの区切り文字を指定する.
skip = 0	最初の行 から読み飛ばす行数を指定する．指定しない場合はファイルの 1 行目から読む.
nrows = −1	何行目まで読むかを指定する ．指定しない場合（負の値の場合）はファイルの終わりまで読む).
header = F	「1 行目は列名が書かれている」か否かを指定する．列名が書かれている場合で，F を指定すると不具合が生じる.
comment.char	コメント行を表す記号（デフォルトは # ）を指定する.
row.names=NULL	1 から始まる番号が行名として生成される.
row.names=”列名”	指定した列のデータが行名として使われる．使われた列はデータフレームの変量としては取り除かれることになる.
row.names=列番号	上記と同じ.
row.names=文字型ベクトル	ベクトルの各要素が行名となる．ただし行数と同じ長さのベクトルを指定する.

また，read.table() のラッパークラスとして以下のような関数が用意されている.

関数	header	sep	quote	dec	用途
read.csv(”ファイル名”)	T	”,”	”¥””	”.”	区切りがコンマの場合
read.csv2(”ファイル名”)	T	”;”	”¥””	”,”	区切りがコロン＆小数点がコンマの場合
read.delim(”ファイル名”)	T	”¥t”	”¥””	”.”	区切りがタブの場合
read.delim2(”ファイル名”)	T	”¥t”	”¥””	”,”	区切りがタブ＆小数点がコンマの場合
read.fwf(”ファイル名”)	F	”¥t”	”¥””	”.”	一行の各欄の桁数 widths を指定して読み込む

例えば data04.txt は関数 read.csv() を用いて読み込むことが出来る.

```
> x <- read.csv("data04.txt")
```

（注意 1）　Windows ユーザーの方は，フォルダの中の『フォルダオプション』から『拡張子を表示しない』のチェックを外した方が良い．チェックを外したくない方はテキストファイルを開いて data という名前で保存する.

（注意 2）　Windows 版 R では，”C:/data.txt” を ”C:¥data.txt” とするとエラーが出るので注意．どうしても ¥ を使いたい場合は ¥¥で ¥を表すことが出来る.

9.2.2　データフレームの作成（EXCEL から）

csv ファイルに保存する方法

目的は関数 read.csv() で読み込める形式にすることである（前節の data04.txt の状態)．まず，EXCEL ファイルを開き，メニューの [ファイル] の [開く] から，[名前をつけて保存] を選択する．保存する名前をつけた後，次に [ファイルの種類] から [CSV カンマ区切り] を選択して保存する．Windows 版 R の場合は以下のようになる.

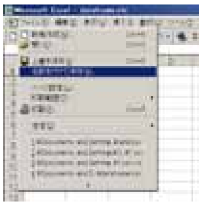


図 9.1　別名で保存

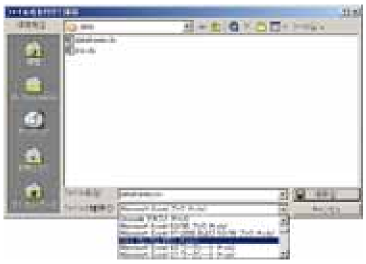


図 9.2　csv 形式で保存

Mac OS X 版 R の場合は以下のようになる．

図 9.3　別名で保存

図 9.4　csv 形式で保存

元の EXCEL ファイルに列名がある場合は単に read.csv(”ファイル名”) とすればよい．列名が無い場合は，関数 read.csv() の引数 header に F を指定し，引数 col.names に列名を指定すれば良い．

```
> ( x <- read.csv("data05.csv", header=F,
+           col.names=c("SEX","HEIGHT","WEIGHT")) )
  SEX HEIGHT WEIGHT
1   F    158     51
2   F    162     55
...................
```

data05.csv

```
F,158,51
F,162,55
M,177,72
M,173,57
M,166,64
```

EXCEL のセルをコピー＆ペーストする方法

まず，EXCEL のセルをコピーする．Windows 版の場合は，列名をコピーしてもしなくてもよいが，Mac OS X 版の場合は列名をコピーしてはいけない（データのみコピー）．

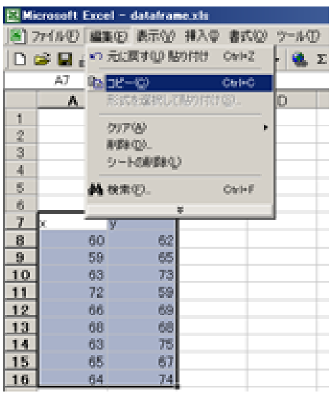


図 9.5　Windows の場合

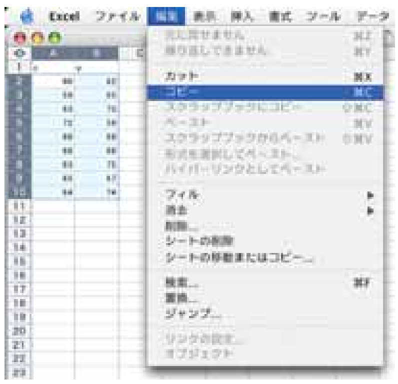


図 9.6　Mac OS X の場合

(1) Windows 版の場合で，列名をコピーした場合は以下のようにすればよい[*4].

```
> x <- read.delim("clipboard")
```

列名をコピーしなかった場合は以下のようにすればよい.

```
> x <- read.delim("clipboard", header=F)
```

(2) Mac OS X 版の場合，まず以下の関数を定義する.

```
excel.mac <- function(...) {
  args <- c(...)
  temp <- matrix(scan(""), byrow=TRUE, ncol=length(args))
  data <- data.frame(temp)
  colnames(data) <- args
  return(data)
}
```

次に，関数 excel.mac(”変数名 1”,”変数名 2”,・・・) を実行する[*4]．すると，以下のような画面になる.

```
> x <- excel.mac("X", "Y")
1:
```

ここで，画面にデータをペーストする．データが読みとられた後，Enter キーを 2 回ほど押すとデータ入力が終了する.

xls ファイルを直接読み込む方法

gregmisc パッケージと Perl[*5]を用意することで, xls 形式のファイルを直接読み込むことが出来る．関数は read.xls() を使い，引数 sheet でシート番号を指定することが出来る.

```
> library(gregmisc)
> x <- read.xls("data01.xls", sheet=1)
```

9.3 データへのアクセス方法

データフレーム x にアクセスする方法を以下に紹介する[*6].

コマンド	機能
x$列名，x[”列名”]，x[[”列名”]]	列データ（例えば SEX や WEIGHT ）にアクセスする.
x[2], x[[2]]	2 番目の列データにアクセスする.
x[3, 2], x[[3, 2]]	3 行 2 列目のデータにアクセスする.
x[[3, ”列名”]], x[[3, ”列名”]]	指定した列の 3 行目のデータにアクセスする.
x[c(1, 2)]	1 列目と 2 列目のデータにのみアクセスする．x[, c(1, 2)] も同じ.
x[c(3, 4),]	3 行目と 4 行目のデータにのみアクセスする.
x[c(1,3,5,2,4),]	順序ベクトル c(1,3,5,2,4) の順に行を並べかえる.

[*4] これは直接コンソール画面に入力すること．R Editor から実行すると，エディタの仕様の関係で上手く機能しない.
ところで，関数 write.table(x, ”clipboard”, sep=”¥t”) でデータ x をクリップボードにコピーすることが出来，この後 EXCEL シートにデータをペーストすることが出来る.

[*5] この章の最後の「落穂ひろい」で Windows 版 ActivePerl のインストール方法を紹介している.

[*6] [ ] でアクセスすると，names 属性付きでアクセスすることが出来る.

コマンド	機能
x[,c(T,F,T)]	論理ベクトル c(T,F,T) が TRUE となっている列にのみアクセスする．
x[x$SEX==”F”,]	性別が F（女性）である行にのみアクセスする．
x[x$SEX==”F” & x$WEIGHT>50,]	性別が F（女性）かつ体重が 50kg 以上である行にのみアクセスする．

以下にいくつかの例を示す．

(1) データの変更や追加，消去方法はベクトルや行列とほぼ同じである．例えば x[[2]] や x[2] でデータフレームの 2 列目にアクセスすることが出来る．

```
> x[[2]]                # 第 2 列のデータを見る -> a[2] でもよい
[1] 158 162 177 173 166
> x[c(1,3,4),]          # 第 1, 3, 4 行を抜き出す
  SEX HEIGHT WEIGHT
1   F    158     51
3   M    177     72
4   M    173     57
```

(2) データを消去する場合は，行や列に NULL を代入すればよい．

```
> x[[2]] <- NULL        # 第 2 列を消去：列の消去は NULL を代入する. x <- x[,-c(2)] でも同じ結果
> x
  SEX WEIGHT
1   F     51
2   F     55
3   M     72
4   M     57
5   M     64
```

9.4　データの結合（マージ）と整列（ソート）

データ加工に関する関数の一覧はこちら（以下に出てくる x と y はデータフレームとする）．

コマンド	機能
ncol(x)	x の列数（項目数）を求める．
nrow(x)	x の行数（データ数）を求める．
names(x)	x の列名を表示する．
rbind(x, y)	x と y を縦に並べて結合する（x と y の列名が全て同じ場合）．
cbind(x, y)	x と y を横に並べて結合する（x と y の行数が同じ場合）．
data.frame(x, y)	x と y を横に並べて結合する（x と y の行数が同じ場合）．
merge(x, y)	x と y を併合（マージ）する．通常は引数に all=T を指定し，データを全て残す．all=T を指定しなければデータの共通部分が結果として返される．

まず，データフレーム D1 と D2 を作成する．

```
> id <- c("A","C","E");     height <- c(158,177,166)
> D1 <- data.frame(ID=id, H=height)
>
> id <- c("A","B","D","E"); weight <- c( 51, 55, 57, 55)
> D2 <- data.frame(ID=id, W=weight)
```

D1

```
  ID   H
1  A 158
2  C 177
3  E 166
```

D2

```
  ID  W
1  A 51
2  B 55
3  D 57
4  E 55
```

(1) 2 つのデータフレームをマージ（併合）する場合は関数 merge() を用いる．関数 merge() は 2 つのデータフレームの両方にある列データ（この場合は ID ）で紐付けをしてデータをマージ（併合）する．このとき，引数 all に何も指定しない場合は ID が共通しているデータのみを残して併合，T を指定した場合は全てのデータが併合される．また，引数 all.x と all.y でそれぞれのデータフレームにおいて，データを残すか否かを指定することが出来る．

```
> merge(D1, D2)
  ID   H  W
1  A 158 51
2  E 166 55
```

```
> ( D <- merge(D1, D2, all=T) )
  ID   H  W
1  A 158 51
2  C 177 NA
3  E 166 55
4  B  NA 55
5  D  NA 57
```

(2) 引数 by で，2 つのデータフレームを紐付けする列名（この場合は ID ）を指定することも出来る．また，異なる列名で 2 つのデータフレームを紐付けする場合は，引数 by.x と by.y で指定すればよい[*7]．さらに，複数の列でデータフレームを紐付けする場合は，引数 by や by.x と by.y に名前のベクトルを指定すれば良い（例：merge(D1, D2, by.x=names(D1), by.y=names(D2), all=T)）．

```
> ( merge(D1, D2, by="ID", all=T) )
  ID   H  W
1  A 158 51
2  C 177 NA
3  E 166 55
4  B  NA 55
........
```

```
> ( merge(D1, D3, by.x="ID",
+          by.y="Num", all=T) )
  ID   H  W
1  A 158 51
2  C 177 NA
3  E 166 55
........
```

D3

```
  Num  W
1   A 51
2   B 55
3   D 57
4   E 55
```

(3) ソートを行う場合は，まず関数 order(昇順の対象となる列) で順番を作成し，それを使ってデータフレームにアクセスすればよい．例として，前々項 (1) でマージしたデータフレーム D の W についてソートを行ってみる．

```
> sortlist <- order(D$W)
> ( D4 <- D[sortlist,] )
  ID   H  W
1  A 158 51
3  E 166 55
4  B  NA 55
5  D  NA 57
2  C 177 NA
```

*7 列名は by.x で指定した名前となる．また，引数 sort に F を指定して自動的に整列するのを制御することも出来る．.

(参考) ソート後は行番号がバラバラになる．行番号ラベルを変更する場合は以下のようにする．

```
> rownames(D4) <- c(1:nrow(D4))
> D4
  ID   H  W
1  A 158 51
2  E 166 55
...........
```

複数の列に対してソートを行う場合は関数 order(1 番目の列, pmax(1 番目の列, 2 番目の列)) として順番を作成し，それを使ってデータフレームにアクセスすればよい．

```
> sortlist <- order(D$W, pmax(D$W, D$H))
> D[sortlist,]  # W を昇順に並べる
  ID   H  W     # W で同じ値がある場合は
1  A 158 51     # H の小さい方を上にする
3  E 166 55
4  B  NA 55
5  D  NA 57
2  C 177 NA
```

```
> sortlist <- order(D$W, pmax(D$W, D$ID))
> D[sortlist,]  # W を昇順に並べる
  ID   H  W     # W で同じ値がある場合は
1  A 158 51     # ID の小さい方を上にする
4  B  NA 55
3  E 166 55
5  D  NA 57
2  C 177 NA
```

9.5　データの加工・抽出

コマンド	機能
head(x, n=a)	先頭から a 行だけ抽出する．関数 tail(x, n=b) で末尾から b 行だけ抽出する．
na.omit(x)	NA を含む行を削除する．関数 na.exclude() も似た様な関数である．
transform(x, y=値)	データフレーム x に新たな列 y を追加する．
x[sapply(x, 論理ベクトル)]	論理ベクトルが TRUE となっている行にのみアクセスする．例えば x[sapply(x, is.numeric)] ならば数値データにのみアクセスする．
subset(x, 条件式)	条件式に合う行のみを抽出する．例えば subset(x, id>3) ならば id > 3 となる行のみを抽出する．
subset(x, 条件式, ベクトル)	ベクトルで指定した列に対し, 条件式に合う行のみを抽出する．例えば subset(x, id>3, c(id,sex)) ならば id > 3 となる ”id” と ”sex” だけを抽出する．
split(x, 列名や条件式)	列がカテゴリーデータならば列名でデータフレームを分割する．条件式でデータフレームを分割することも出来る（例：split(x, x$w==70)）．
by(x, x$列名, mean)	データ x を層別して平均を求める．mean を別の関数に変えても良い．関数 aggregate() も似た機能を持つ関数である．
reshape(x, ...)	データフレームを横に展開する．

まず，データフレーム DF を作成する．

```
> id  <- c( 1, 2, 3, 4, 5)                  # ID
> sex <- c("F","F","M","M","M")             # 性別（F:女，M:男）
> h   <- c(158,162,177,173,166)             # 身長
> w   <- c( 51, 55, 72, 57, 64)             # 体重
> DF  <- data.frame(ID=id, SEX=sex, H=h, W=w)
```

DF

```
  ID SEX   H  W
1  1   F 158 51
2  2   F 162 55
3  3   M 177 72
4  4   M 173 57
5  5   M 166 64
```

(1) データフレーム DF の体重（W）の列だけを足し算したい場合は以下のように列ラベルを指定し，それごと関数 sum() に放り込んでやればよい．また，体重の単位を g（グラム）に変換する場合は，体重（W）の列を 1000 倍したものをデータフレームに代入してやればよい．

```
> sum(DF$W)                         # 体重の和を求める
> DF$W <- DF$W * 1000               # 体重の単位を kg から g に変換する
> DF
  ID SEX   H     W
1  1   F 158 51000
2  2   F 162 55000
..................
```

DF に新たな変数 G を追加する場合は，関数 transform() を使うか，単に代入すればよい．

```
> transform(DF, G=DF$W * 1000 )     # DF$G <- DF$W * 1000 でも可
  ID SEX   H  W     G
1  1   F 158 51 51000
2  2   F 162 55 55000
.....................
```

(2) 条件に当てはまる行のみに処理を施す場合は，関数 ifelse(条件式, TRUE の場合に返す値, FALSE の場合に返す値) を用いる．

```
> DF$W <- ifelse(DF$SEX=="F", NA, DF$W)  # 女性の体重を隠す（NA に）
> DF
  ID SEX   H  W
1  1   F 158 NA
2  2   F 162 NA
3  3   M 177 72
...............
```

(3) 条件に当てはまるものだけを抽出する場合は，条件式をデータフレームの [ ] などで指定すればよい．

```
> cond <- (DF$H >= 170)             # 身長（H）が 170cm 以上の人を抽出
> DF[cond,]
  ID SEX   H  W
3  3   M 177 72
4  4   M 173 57
```

関数 subset() や関数 sapply() を用いても，条件に当てはまるものだけを抽出することが出来る．

```
> subset(DF, ID>3)                  # ID > 3 となる行
  ID SEX   H  W
4  4   M 173 57
5  5   M 166 64
> subset(DF, ID>3, c(ID, SEX))      # ID > 3 となる "ID" と "SEX" のみ
  ID SEX
4  4   M
5  5   M
```

(4) 整数データをカテゴリ変数と認識させる場合は関数 as.factor() で変換すればよい[*8].

```
> DF$ID <- as.factor(DF$ID)        # データを見ても変換されたかどうかが分からない
> DF[sapply(DF, is.numeric)]       # 数値変数だけを選択
    H  W
1 158 NA
2 162 NA
........
```

(5) ID を固定し，SEX（性別）で場合分けしてデータを横に展開する場合は関数 reshape() を用いる．

```
> reshape(DF, timevar="SEX", idvar="ID", direction="wide")
  ID H.F W.F H.M W.M     # H.F：女性の身長，W.F：女性の体重，H.M：男性の体重，W.M：男性の体重
1  1 158  51  NA  NA
2  2 162  55  NA  NA
3  3  NA  NA 177  72
4  4  NA  NA 173  57
5  5  NA  NA 166  64
```

9.6 データの編集・他の型への変換

(1) データをセル形式で見る場合は関数 edit(データフレーム名) を用いる[*9].

```
> DF <- edit(DF)
```

データをクリックすることで値を編集することが出来，列名をクリックすれば数値型と文字型の選択，列名の変更などが行える．編集が終了したらウインドウを閉じることで，変更が反映される．

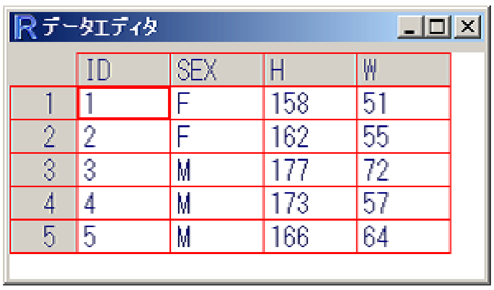


図 9.7　データフレームの編集（Windows）

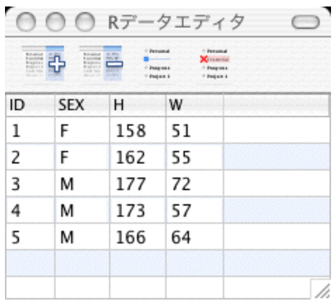


図 9.8　データフレームの編集（Mac OS X）

(2) データフレームを行列などの他の型に変換する場合は，62 頁に紹介されている関数を用いればよい．例えば，データフレームを行列に変換する場合は関数 data.matrix() を用いる．また，データフレームに関する詳細な情報を得る場合は関数 str() を，データフレームかどうかを検査する場合は関数 is.data.frame() を用いればよい．

[*8] 「順序変数宣言を行う場合は as.ordered(A$id)」「量的変数宣言を行う場合は as.single(A$w)」とすればよい．

[*9] 以下に類似の関数を示す．例えば data.entry(DF$ ID, DF$ SEX) と，列の一部を編集したいときに使用する．

data.entry(..., Modes = NULL, Names = NULL) ：　Modes で変数に使用されるモード，Names で変数に使用される名前を指定してセルで編集する．

dataentry(data, modes) ：　data で数値および文字ベクトルのリストを，modes で変数のモードを与えるデータの長さのリスト (list() でも可) を指定してセルで編集する．

de(..., Modes = list(), Names = NULL) ：　上の 2 つとほぼ同様．

9.7 落穂ひろい

9.7.1 組み込みデータにアクセスする方法

R には多数のデータセットが用意されている．これは関数 data() を用いることで一覧を見ることが出来る．

```
> data()                         # 全てのデータセットを表示
> data(package ="base")          # base パッケージのデータセットを表示
> data(USArrests, "VADeaths")    # データ USArrests，VADeaths をロード
> help(USArrests)                # データ USArrests のヘルプを見る
```

もしデータがライブラリの中に入っている場合は，先にライブラリを呼び出す必要がある．

```
> library(nls)                  # ライブラリ nls の呼び出し
> data(Puromycin)               # データ‘Puromycin’の呼び出し
```

9.7.2 attach() と detach()

データフレームの要素を参照するときは『データフレーム名$要素のグループ』と $ を用いて参照する．しかしデータフレーム名をいちいち指定するのは面倒である．データフレームの成分を，リスト名を明示的に指定することなしに指定できれば楽である．例えば myframe が myframe$u，myframe$v，myframe$w を持つデータフレームであるとする．このとき関数 attach(myframe) を実行することで，myframe$ を付けすに u，v，w だけで変数にアクセスすることが出来るようになる．ただし，直前までに u という名前の変数を使っていると不具合が出るので注意が必要である．

```
> u <- 1:5;  v <- rnorm(5);  w <- u*v
> myframe <- data.frame(u=u, v=v, w=w)
> attach(myframe)
> w              # attach() を実行することで列名 w だけでデータにアクセス出来る
[1]  0.166959  1.676896  1.435630 -3.024536  1.859002
```

関数 attach() は組み込みデータを使う際にも有用である[*10]．

```
> data(sleep)     # R に標準で用意されているデータ（W.S.Gosset（Student）の睡眠薬の比較データ）
> attach(sleep)
> extra           # attach() を実行することで extra だけでデータにアクセス出来る
 [1]  0.7 -1.6 -0.2 -1.2 -0.1  3.4  3.7  0.8  0.0  2.0  1.9  0.8
[13]  1.1  0.1 -0.1  4.4  5.5  1.6  4.6  3.4
```

データフレームを検索リストから外すには関数 detach() を用いればよい．これ以後，myframe$u の値を参照するには，u だけを入力しても参照することは出来ない．関数などの繰り返し文中等で同じデータフレームを何度も attach

[*10] 関数 attach(myframe) を実行することで，リストの成分をリスト名を明示的に指定することなしに指定することも出来る．

```
> myframe <- list(c("a","b","c"), c(1:3), c(1.5,3.2,-0.8))
> names(myframe) <- c("u", "v", "w")
> attach(myframe)          # attach() を実行することで
> u                        # u だけでデータにアクセス出来る
[1] "a" "b" "c"
```

する場合は，一まとまりの処理が終って不要になったらこまめに detach() (例：on.exit(detach(sleep)) を関数本体に入れる) しないと，処理時間が次第に遅くなるとので注意.

```
> search()          # どんなリストやデータフレームがあるかを調べる
> detach(sleep)     # sleep を外す
```

処理時間の問題が気になる場合は，attach() に代わる関数 with() を用いる．with() はデータフレームなどから専用の環境を作ってその中で作業を行う．結果として attach() せずに成分を参照することが出来，with() の作業が終了すればこの環境もなくなるので，変数名の衝突などの不具合が起きない.

```
> data(sleep)
> with(sleep, {                  # sleep から専用の環境を作ってその中で F 検定を行う
+   x <- extra[group==1];  y <- extra[group==2];  var.test(x, y)
+ })
      F test to compare two variances（結果の出力は省略）
```

9.7.3　R 以外のソフトで作成されたデータファイルの読み込み

R 以外の統計ソフト (SAS など) で作成されたデータファイルからデータを読む場合は，パッケージ foreign 中の関数を用いる．例えば SPSS のデータファイルを読み込む場合は以下のようにする.

```
> library(foreign)
> read.spss("datafile", use.value.labels=FALSE)
```

R の foreign パッケージには以下の関数が用意されている.

関数	機能
lookup.xport()	SAS XPORT フォーマットライブラリについての情報を探す.
read.dta()	Stata binary files を読み込む.
read.epiinfo()	Epi Info data files を読み込む.
read.mtp()	Minitab Portable Worksheet を読み込む.
read.spss()	SPSS data file を読み込む.
read.ssd()	read.xport() によって SAS のパーマネント・データセットからデータフレームを得る.
read.xport()	SAS XPORT format library (ver.6) を読み込む.
write.dta()	Stata binary format をファイルに書き出す.

また，gregmisc パッケージの関数 read.xls() を使えば，EXCEL ファイル（.xls）を直接読み込むことが出来る．ただし，Windows 版 R の場合で Perl を入れる必要がある．以下に，ActiveState：*http://www.activestate.com/* から「ActivePerl」を入れる方法を紹介する．まず，図 9.9 の「ActivePerl x.x.x.xxx」をクリックし，図 9.10 の「Download」をクリックする.

図 9.9

図 9.10

すると，ユーザー名を入れる画面になるので，（名前を入れたい人は名前を入れてから）Next> をクリックする．すると，ダウンロード出来る実行ファイル一覧が出るので，Windows 版の ActivePerl をダウンロードする．OS が 2000/XP の方は MSI 版をダウンロードしてインストール，98/Me/NT の方は図 9.12 の真ん中あたりにある「Windows Installer」をインストールしてから MSI 版をダウンロードしてインストールすればよい．

図 9.11

図 9.12

ActivePerl をインストールする場所は変更しない方がよい（デフォルトは C:¥perl¥bin¥perl.exe ）．変更すると，関数 read.xls() を実行する際，その都度 ActivePerl の実行ファイルがある場所を引数 perl で指定しなければいけなくなる．

9.7.4　欠損の扱い

手入力でデータフレームを作成する場合で欠損が含まれているデータを読み込む場合は，ベクトル中の欠損部分を NA としておけば，該当部分に欠損値（NA）が入る．

```
> sex <- c("F",NA,"M"); height <- c(158,162,NA); weight <- c(51,55,72)
> ( x <- data.frame(SEX=sex, HEIGHT=height, WEIGHT=weight) )
   SEX HEIGHT WEIGHT
1    F    158     51
2 <NA>    162     55
3    M     NA     72
```

ファイルからデータを読み込む場合で欠損が含まれているデータを読み込む場合は，data06.txt のように，データ間がコンマ（ , ）で区切られている方が処理しやすい．この場合，単に欠損部分を空白にしておけば，該当部分に欠損値（NA）が入る．

```
> ( x <- read.table("data06.txt", header=T, sep=",") )
  sex height weight
1   F    158     51
2   F    162     55
3   M     NA     72
4   M    173     57
5   M    166     64
```

data06.txt

```
sex,height,weight
F,158,51
F,162,55
M,   ,72
M,173,57
M,166,64
```

9.7.5　scan() を用いてファイルからベクトルデータを読み込む

関数 scan() について

関数 scan(ファイルのパス) を用いると，ファイルからベクトルとしてデータを読み込むことが出来る．*11．最も簡単にデータ（例えば右下の data07.txt ）を読み込むには以下のようにすれば良い．以下では全 6 行のうち 1 行目を読み飛ばして x にデータ (データフレーム) を読み込んでいる．

```
> x <- scan("data07.txt", skip=1, nlines=6)
Read 20 items
> x
 [1] 171.0  62.6  23.0   1.0 158.0  49.8
 [7]  25.0   0.0 163.0  68.7  42.0   1.0
[13] 178.0  75.4  33.0   1.0 163.0  55.7
[19]  28.0   0.0
```

data07.txt

```
# data.txt
171 62.6 23 1
158 49.8 25 0
163 68.7 42 1
178 75.4 33 1
163 55.7 28 0
```

関数 scan() で読み込んだデータはベクトルデータである．これを行列に変換する場合は以下のようにすればよい．

```
x <- matrix( scan("data07.txt", skip=1, nlines=6), 5, byrow=T)
```

list（ダミーのリスト）を用いると，5 行 4 列のデータとして読み込むことが出来る上，該当する列の ”” にラベルを入れることでラベルをつけることが出来る．以下はデータの型が全て文字型であると R に指定していることになる．

```
> x <- scan("data07.txt", list("",weight="","",sex=""),skip=1, nlines=6)
Read 5 records
> x
[[1]]
[1] "171" "158" "163" "178" "163"
$weight
[1] "62.6" "49.8" "68.7" "75.4" "55.7"
[[3]]
[1] "23" "25" "42" "33" "28"
$sex
[1] "1" "0" "1" "1" "0"
```

*11 R でファイルからデータを読み込むのは関数 read.table() の方が便利であると思われるが，scan() よりもメモリを多く必要として実行速度も scan() より劣る．本格的に多くのデータを取り込む場合，複雑な規則でデータの読み込みを行いたい場合には scan() を使用した方が良い．また，データの出力 は，関数 read.table() に対して関数 write.table() や write() で実現できる．

データの型を数値であると指定する場合は ラベル = 0 (または ラベル = 1 など) とすればよい．以下では sex は性別なので文字型，それ以外は数値として読み込んでいる．

```
> x <- scan("data07.txt", list(h=0,w=1,a=0,s=""), skip=1)
Read5 records
> x
$h
[1] 171 158 163 178 163
$w
[1] 62.6 49.8 68.7 75.4 55.7
$a
[1] 23 25 42 33 28
$s
[1] "1" "0" "1" "1" "0"
```

このときラベルを付けない場合は R の方で勝手にラベルをつける．以下では sex は性別なので文字型，それ以外は数値として読み込んでいる． あとは read.table() で読み込んだのと同様にしてアクセスすることが出来る．また，edit() を使って編集することも出来る．

```
> x$h <- x$h/100
```

以下に関数 scan() の引数を，関数 read.table() の引数と併せて紹介する．

関数 read.table() の引数

引数	説明
sep = ””	データとデータの区切り文字を指定する．
skip = 0	最初の行 から読み飛ばす行数を指定する．指定しない場合はファイルの 1 行目から読む.
nrows = −1	何行目まで読むかを指定する ．指定しない場合（負の値の場合）はファイルの終わりまで読む).
header = F	「1 行目は列名が書かれている」か否かを指定する．列名が書かれている場合で，F を指定すると不具合が生じる．
comment.char	コメント行を表す記号（デフォルトは ＃ ）を指定する．
その他	quote = ””，row.names, col.names などがある．

関数 scan() の引数

引数	説明
sep = ””	データとデータの区切り文字を指定する．
skip = 0	最初の行から読み飛ばす行数を指定する．指定しない場合はファイルの 1 行目から読む.
nlines	何行目まで読むかを指定する．指定しない場合はファイルの終わりまで読む.
what	データのタイプを指定する．
nmax = −1	データを読み込む個数を指定する．nmax について，what にリストが指定されていたらリストを読み込む個数を指定することが出来る．
その他	quote = ””，n = −1，na.strings = ”NA” などがある．

関数 scan() の使用上の注意

関数 scan() で読み込む際，最後は改行文字で終わっていなければいけない．例えば，データが 2 行ある場合は，2 行目の最後で必ず改行しないと警告が出る．

```
> x <- scan("data08.txt", sep=",")
Read 10 items
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

data08.txt

```
1,2,3,4,5
6,7,8,9,10
```

行の最後は改行のみとすること．例えば data09.txt のように区切り文字がカンマ（ , ）の場合で，行の最後にカンマが入っている場合は，結果に不要な欠損値が入ってしまう．

```
> x <- scan("data09.txt", sep=",")
Read 11 items
> x
 [1]  1  2  3  4  5 NA  6  7  8  9 10
```

data09.txt

```
1,2,3,4,5,
6,7,8,9,10
```

data10.txt

```
1,2,3,4,5,
6,7,8,9,10,
```

9.7.6　その他の注意

- データに ” や ’ が出てくると，対応する引用符までひとつの文字列として読み込まれる．
- 未充足の行がある場合は fill = TRUE とすること．
- もし分離記号が指定されているならば，文字欄の先頭，空白が続いている箇所は欄の一部とみなされてしまう．空白を取り去るには引数に strip.write = TRUE と指定すればよい．
- read.line() では空行は無視される．blank.lines.skip = FALSE とすればこれを欠損として読み込ませることが出来る． fill = TRUE と組み合わせることも出来る．

第 10 章

ファイルへのデータ出力

10.1　データフレームの出力

read.table() に対して関数 write.table() や write() で実現できる[*1]．例えば以下の様な 5 行 3 列のデータが入ったファイル data07.txt が（ C:/ に）あったとする．このデータ data.txt を関数 read.table() で読み込んで，関数 write.table(データフレーム名, "出力するファイルのパス") でファイル output.txt に出力してみる．write.table(x,"C:/output.txt") だけでは（quote=F にしないと）要素に "" がついてしまう．

```
> x <- read.table("C:/data11.txt")
> write.table(x, "C:/output.txt", quote=F,
              col.names=F, append=T)

  # append=F にするとデータをファイルに上書きする
```

data11.txt

```
0.9, 1.2, 1.9
1.3, 1.6, 2.7
2.0, 3.5, 3.7
1.8, 4.0, 3.1
0.9, 1.2, 1.9
```

データフレームを出力する際は write.table() が使いやすいが，リストをファイルに出力する場合は write(リスト名, "出力するファイルのパス") が使いやすい．それぞれの関数の書式は以下のようになっている．「出力するファイルのパス」は，空文字列 "" なら標準出力に出力される．

- write(リスト名, "出力するファイルのパス", append = T, ncolumns = 一行の要素の数)
- write.table(データフレーム名, "出力するファイルのパス", append=T, quote=F, col.names=F)

関数 write() について使い方の例を以下に挙げる．

```
> x <- matrix(1:8, ncol=2)              # 4 * 2 の行列
> write(x, file="data12.txt")           # ファイル data12.txt に書き出し
> scan("data12.txt")                    # scan 関数で再読み込み（結果はベクトル）
Read 8 items
[1] 1 2 3 4 5 6 7 8
> y <- matrix(scan("data12.txt"), ncol=4)  # また行列に変換
Read 8 items
> y
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

[*1] MASS ライブラリ中の関数 write.matrix() も write.table() と同様の機能を持つ．

ファイルそのものを直接編集しない場合，関数 save() でデータの構造をそのまま記録してくれる．呼び出す場合は関数 load() を用いる．

```
> save(x, file="data13.dat")
> rm(x)                                    # ファイル data13.dat は
> load("data13.dat")                       # テキストデータではない！
```

10.2　区切り文字を付けたデータの出力

リストに入ったデータを，例えばカンマ , で区切ってファイルに出力する場合，まず，{ データ , カンマ，データ , カンマ，··· } というリストを作っておいて，次に関数 write() で 1 行ごとにファイルに出力すればよい．

```
> x <- c(1:9)
> out <- NULL
> for (i in 1:(length(x)-1)) {
+   out <- cbind(out, x[i])
+   out <- cbind(out, ",")
+ }
> out <- cbind(out, x[length(x)])
> write(out, file="C:/data14.txt",
        ncolumns = 2*length(x))
```

data14.txt

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

また，関数 writeLines(paste(データ), sep=”,”) とすれば区切り文字を綿密に指定してファイルに書き込みを行うことが出来る．この際，明示的に改行 ¥n を指定しないと改行されない．

```
> out <- file("C:/data15.txt", "w") # 書き込みモードで開く
> for (i in 1:10) {
+   if (i < 5)       writeLines(paste(i), out, sep=", ")
+   else if (i == 5) writeLines(paste(i), out, sep="\n")
+   else if (i < 10) writeLines(paste(i), out, sep=", ")
+   else             writeLines(paste(i), out, sep="")
+ }
> close(out)                          # ファイルを閉じる
```

data15.txt

```
1, 2, 3, 4, 5
6, 7, 8, 9, 10
```

10.3　出力したファイルを EXCEL で読み込む

R で出力したファイルを EXCEL から読み込む場合の一つの例を示す．

(1) 前節の説明の方法で，データをカンマ区切りで出力し，拡張子を【.csv】として保存する（【.txt】としても読み込むことは出来る）．
(2) メニューの [ファイル] の [開く] から，出力されたファイルを開く．その際，ファイルの種類を [テキスト ファイル (∗.prn, ∗.txt, ∗.csv)] とする．
(3) 目的のファイルを選択して 開く をクリックする．

10.4 データを LaTeX 形式，html 形式で出力

データを LaTeX の表を出力する命令に書き直す場合は，パッケージ xtable の関数 xtable() を用いる．

```
> library(xtable)
> mytable <- xtable( matrix(1:24,4,6) )
> print(mytable)
% latex table generated in R 2.1.0 by xtable 1.2-4 package
% Sat May 28 14:47:23 2005
\begin{table}[ht]
\begin{center}
\begin{tabular}{rrrrrrr}
\hline
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 1.00 & 5.00 & 9.00 & 13.00 & 17.00 & 21.00 \\
2 & 2.00 & 6.00 & 10.00 & 14.00 & 18.00 & 22.00 \\
3 & 3.00 & 7.00 & 11.00 & 15.00 & 19.00 & 23.00 \\
4 & 4.00 & 8.00 & 12.00 & 16.00 & 20.00 & 24.00 \\
\hline
\end{tabular}
\end{center}
\end{table}
```

また，データを html の表を出力する命令に書き直す場合は以下のようにする[*2]．

```
> print(mytable, type="html")
<!-- html table generated in R 2.1.0 by xtable 1.2-4 package -->
<!-- Sat May 28 14:48:09 2005 -->
<TABLE border=1>
<TR><TH></TH><TH> 1 </TH><TH> 2 </TH><TH> 3 </TH><TH> 4 </TH><TH> 5 </TH><TH> 6 </TH>
</TR><TR><TD align="right"> 1 </TD><TD align="right"> 1.00 </TD><TD align="right"> 5.00 </TD>
<TD align="right"> 9.00 </TD><TD align="right"> 13.00 </TD><TD align="right"> 17.00 </TD>
<TD align="right"> 21.00 </TD></TR>
<TR><TD align="right"> 2 </TD><TD align="right"> 2.00 </TD><TD align="right"> 6.00 </TD>
<TD align="right"> 10.00 </TD><TD align="right"> 14.00 </TD><TD align="right"> 18.00 </TD>
<TD align="right"> 22.00 </TD></TR>
<TR><TD align="right"> 3 </TD><TD align="right"> 3.00 </TD><TD align="right"> 7.00 </TD>
<TD align="right"> 11.00 </TD><TD align="right"> 15.00 </TD><TD align="right"> 19.00 </TD>
<TD align="right"> 23.00 </TD></TR>
<TR><TD align="right"> 4 </TD><TD align="right"> 4.00 </TD><TD align="right"> 8.00 </TD>
<TD align="right"> 12.00 </TD><TD align="right"> 16.00 </TD><TD align="right"> 20.00 </TD>
<TD align="right"> 24.00 </TD></TR></TABLE>
```

[*2] 他にもパッケージ Hmisc 中の関数 latex() などがある．

第Ⅴ部

グラフィックス篇

第 11 章

作図の準備

今すぐ作図を行いたいという方は，この節は読み飛ばして関数 plot() の章へ読み進んで下さい．

11.1　作図を行うには

R の作図機能は極めて多彩である．さまざまな種類の統計グラフを描述することが出来，全く新しい種類のグラフを描くことも出来る．R で作図を行なうには作図関数，作図デバイスの 2 つを用いて図を描く．

作図関数 ：　実際に何かを描画する関数
作図デバイス ：　図を出力する装置 (デバイス)*1

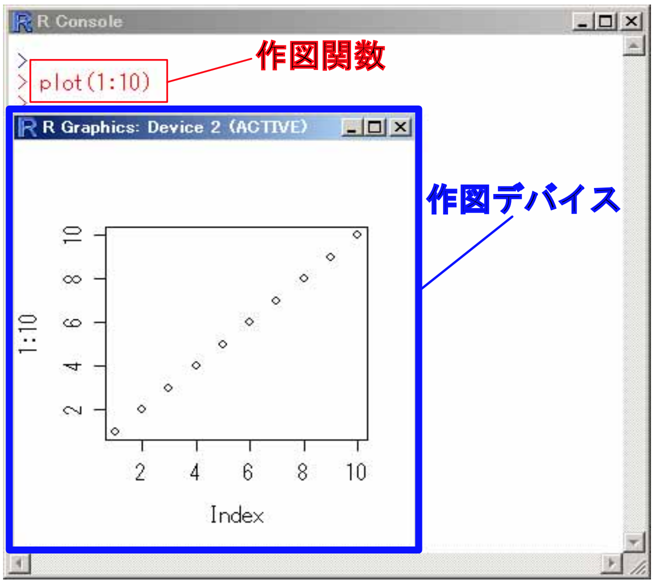


図 11.1　作図関数と作図デバイス

11.2　作図関数について

作図関数は出力装置に依存せず，例えば X ウインドウ上へ描画する場合でも，ポストスクリプト形式で作図出力を得る場合でも，同じ関数を使って同じ手順で作図が行なえるようになっている．作図命令は以下の 3 つの基本的なグ

*1 作図関数は high (low) level plotting function と graphics function の 2 通りの英語があるので困るのだが，作図関数と作図デバイスはそれぞれ，グラフィックス関数やグラフィックスデバイスという訳が適切なのかもしれない · · · ので，適宜読み替えて頂きたい．ただし，パラメータに関しては RjpWiki の記事にあったので，グラフィックスパラメータと訳している．

ループに分けられる．

高水準作図関数： 一枚の完成された図を作成するための関数で，例えば散布図や関数，ヒストグラムを描く．

低水準作図関数： 高水準作図関数で描いた図に追記するための関数で，線や点，文字列や多角形の塗りつぶしを行ったり，座標軸を描く，タイトルを記入するといった図の一部分を描いたりする．

対話的作図関数： マウスなどで図表からプロット点を追加したり除いたりするための関数で，ウインドウ上に表示される "+" マーク (クロスヘア) をマウスで操作し，位置を決めて点や文字を描く．

11.3 作図デバイスについて

R はいろいろな作図関数を備えており，作図結果を様々な出力装置から得ることが出来る．出力する装置のことを R では作図デバイスあるいは作図機器と呼ぶ．R が起動されたとき，例えば UNIX , Linux では自動的に関数 X11() で，Windows では関数 windows() でデバイスドライバが初期化され，デバイスが稼動する．一度デバイスが稼動し始めれば，R の作図命令で様々なグラフ表示や自分でグラフ表示を定義することが出来るようになる．通常，作図デバイスは R が起動されたときに自動的に呼び出されるので，グラフィックスを行う際に作図デバイス云々を気にする必要はほとんどない．

R がサポートしている作図デバイスを呼び出す関数は help(”Devices”) で調べることが出来るが，代表的なデバイス関数には以下のようなものがある．

デバイス	説明	デバイス	説明
bitmap()	ビットマップ	bmp()	ビットマップ
dev2bitmap()	ビットマップ	jpeg()	JPEG
pdf()	ADOBE PDF	pictex()	LaTeX ファイル
png()	PNG	postscript()	POSTSCRIPT
quartz()	(Mac OS 版のデフォルトドライバ)	windows()	(Windows 版のデフォルトドライバ)
win.graph()	(Windows 版のドライバ)	win.metafile()	Windows 版のメタファイル形式
win.print()	(Windows 版のドライバ)	x11()	(UNIX のデフォルトドライバ)
X11()	(UNIX のデフォルトドライバ)	xfig()	Xfig

あるデバイスの利用が終了した場合に dev.off() でデバイスを閉じることも出来る．例えば postscript() でポストスクリプトファイルを作成する場合，ファイルが壊れる不具合を防ぐには dev.off() でデバイスをすぐに閉じればよい．

```
> data(cars)                                              # PS デバイスを開き，出力 file を指定
> postscript("myplot.eps", horizontal=FALSE, height=9, # horizontal = FALSE を指定しないと
             width=14,    pointsize=15)                   # TeX に取り込んだ際，横にひっくり返る
> plot(cars, main = "Speed and Stopping Distances of Cars")
> dev.off()          # 必要な出力がすべて終ったらすぐにデバイスを閉じると不具合が起こりにくい
null device
          1
```

(注) 作業ディレクトリのフルパスを指定しない限り，画像ファイル (この場合は myplot.eps) は現在の作業ディレクトリに保存される．作業ディレクトリの参照・変更方法は 22 頁の「作業ディレクトリの変更」を参照のこと．

11.4　複数のデバイスドライバ

同時に複数の作図デバイスを持つということがある．もちろん一度に一つの作図デバイスだけが作図命令を受け入れることが出来る (このデバイスをカレントデバイスという) のだが，複数のデバイスが開かれているときは，それらは待機状態としてデバイスリストを成していることになる．複数のデバイスを操作するための主要な命令は以下の通りである．

デバイス	説明
デバイス関数 (bmp()，jpeg()，pdf() など)	新しい作図デバイスを開いて待機状態を表すデバイスリストを更新し，現在のデバイス (カレントデバイス) に指定する．今後はこのデバイスにグラフ出力が送られることになる．
dev.list()	アクティブなデバイスの番号と名前を返す．
dev.next()	カレントデバイスの次の位置の作図デバイスの番号と名前を返す．
dev.prev()	カレントデバイスの前の位置の作図デバイスの番号と名前を返す．
dev.set(which=k)	デバイスリストの k 番目の作図デバイスの番号と名前を返す．
dev.off(k)	デバイスリストの k 番目の作図デバイスを終了させる．
dev.copy(device, which=k)	作図デバイス k のコピーを作成する．device は postscript などのデバイス関数を引数にする．
dev.copy2eps(file=””)	eps 出力を指定した dev.copy() と同じ動作を行う．様々なフォント名を指定することが出来る．
dev.print(device,which=k)	アクティブな作図デバイス k のコピーを作成し，他のグラフィックスデバイス (既定では postscript) に再出力する．複製されたデバイスは直ちに閉じられ，終了処理がなされる．
graphics.off()	無効なデバイスを取り除き，リスト中の全てのデバイスを終了する．

以下に，図を PDF ファイルに保存する例を挙げる．後者の方法ならば，png() , bmp() , jpeg() を指定することで，それぞれの形式の画像ファイルが得られる．

```
> plot(1:10)
> dev.copy(pdf, file="finename.pdf")
> dev.off()                              # 必要な出力がすべて終ったらすぐにデバイスを閉じる
>                                        # または，前述の作図デバイスで同じことをやると ...
> pdf()                                  # pdf デバイスを開く ()
> plot(1:10)                             # プロット -> Rplots.pdf に出力
> dev.off()                              # 必要な出力がすべて終ったらすぐにデバイスを閉じる
```

また，図を eps 形式で保存する場合は以下のようにすればよい．

```
> X11()                                  # X11 デバイスを開く（ OS によっては省略可）
> plot(1:10)
> dev.copy2eps(file="finename.eps", width=6) # 幅と高さの一方を省略してもよい
>                                        # または... dev.print() でコピー出力すると...
> X11()                                  # X11 デバイスを開く（ OS によっては省略可）
> plot(1:10)                             # X11 デバイスに出力
> dev.print(file="finename.eps",         # 既定で EPS ファイルに出力
          width=10, height=10, horizontal=FALSE)
```

Windows 版 R ならば，図を右クリックしてメニューから出力することが出来る．メニューの機能は上から『 metafile としてクリップボードにコピー』『 bmp としてとしてクリップボードにコピー』『 metafile (.emf) 形式で保存』『 postscript (.ps , .eps) 形式で保存』『プリントアウト』となっている．

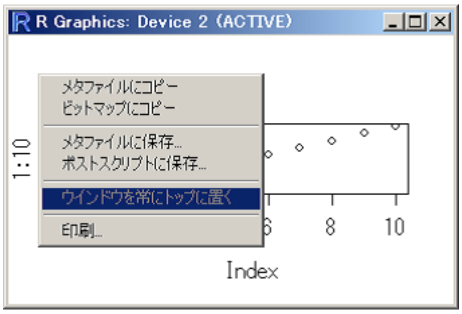


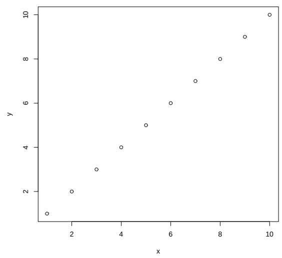
図 11.2　Windows 版 R の右クリックメニュー

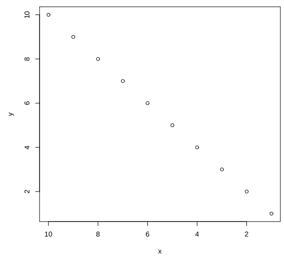
11.5　とりあえず plot()

R で一番良く使われる高水準作図関数が関数 plot() である．最も基本的で機能も多い関数も plot() である．この関数を使って散布図や折れ線グラフなどを描くことが出来る．例えばデータが入っているベクトル x，y を点の座標として以下の様に入力する．

```
> x <- 1:10
> y <- 1:10                   # plot(x 軸のデータ, y 軸のデータ, オプション)
> plot(x, y)                  # 範囲は自動で決まる (xlim=c(1,10) を指定した場合と同じ)
> plot(x, y, xlim=c(10,1)) # x 軸の正の向きを左向きにすることも出来る
```

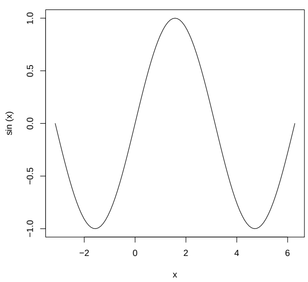
すると散布図の出力が得られる．プロット範囲は引数 xlim, ylim で決めることが出来る．

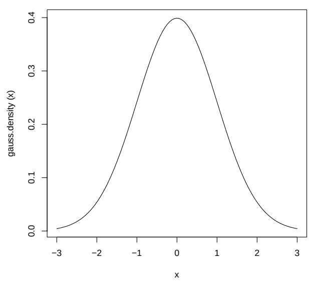




また，数学関数を与えてそのグラフを出力することも出来る．

```
> plot(sin, -pi, 2*pi)                                # plot(関数名, 下限, 上限)
> gauss.density <- function(x) 1/sqrt(2*pi)*exp(-x^2/2)  # 標準正規分布の密度
> plot(gauss.density,-3,3)
```



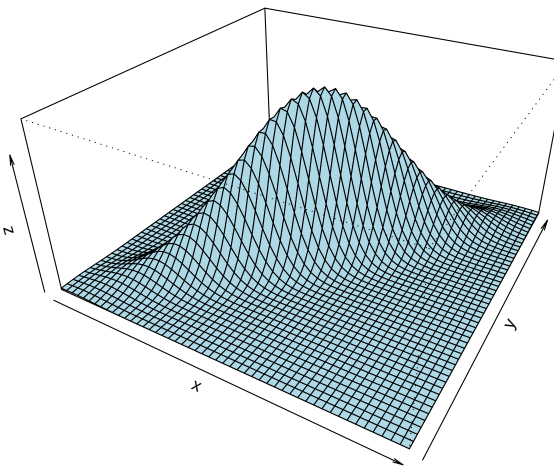


ちなみに，3 次元的にプロットする場合は関数 persp() を用いる．

```
> persp(x 軸のデータ, y 軸のデータ, z 軸のデータ, col = 色,
+       theta = 横回転の角度, phi = 縦回転の角度, expand = 拡大率, border=NA)
```

例として 2 次元正規分布を描く．

```
> x <- seq(-3,3,length=50)      # x 方向の分点
> y <- x                        # y 方向の分点
> rho <- 0.9                    # 2 次元正規分布の定数
> gauss3d <- function(x,y) {    # 2 次元正規分布の関数
+   1/(2*pi*sqrt(1-rho^2))*exp(-(x^2-2*rho*x*y+y^2) / (2*(1-rho^2)))
+ }
> z <- outer(x,y,gauss3d)       # 外積をとって z 方向の大きさを求める
> z[is.na(z)] <- 1              # 欠損値を 1 で補う
> persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue")
```



11.6　plot() の形式指定

関数 plot() の引数 type によって様々な形式でプロットすることが出来る．type には次の 9 種類の文字のいずれかを指定することが出来る．

引数	機能
type=”p”	点プロット (デフォルト)
type=”l”	線プロット (折れ線グラフ)
type=”b”	点と線のプロット
type=”c”	”b” において点を描かないプロット
type=”o”	点プロットと線プロットの重ね書き
type=”h”	各点から x 軸までの垂線プロット
type=”s”	左側の値にもとづいて階段状に結ぶ
type=”S”	右側の値にもとづいて階段状に結ぶ
type=”n”	軸だけ描いてプロットしない (続けて低水準関数でプロットする場合)

以下に例を示す.

```
> x <- rnorm(10)
> plot(x, type="l")
```

11.7 対数軸や軸の範囲の指定

軸に関する設定を行なう為に以下の引数を追加することが出来る.

引数	機能
log=”x”	”x”（ x 対数軸），”y”（ y 対数軸），”xy” (両対数軸) の何れかを指定することが出来る (対数は常用対数のみ)．
xlim=c(0, 1)，ylim=c(−1, 1)	長さ 2 のベクトルで x 座標， y 座標の最小値と最大値を与える，他にも xlog, ylog で対数プロットが出来る．ベクトルを降順に並べる (例：c(2, −2)) と，プロットの向きが逆になる.
axes=FALSE	軸の生成を抑制する．軸の他に表題，刻み，目盛も描くかどうかを論理値で指定する（省略時は TRUE)．他に xaxs, yaxs が指定出来る.

以下に例を示す.

```
> x <- rnorm(10)
> plot(x, ylim=c(-30, 30), type = "l")
```

11.8 タイトルなどの指定

表題などに関する設定を行なう為に以下の引数を追加することが出来る.

引数	機能
main=”Title”	タイトルを与える文字列を指定する．この引数を省略するとは表題は描かれない.
sub=”SubTitle”	サブタイトルを与える文字列を指定する．この引数を省略すると副題は描かれない.
xlab=”X-Label”，ylab=”Y-Label”	それぞれ x 座標名， y 座標名を与える文字列を指定する．省略すると，x 軸のデータとして与えられた引数の名前が座標名として描かれる.
ann = F	軸のラベルを描かないようにすることも出来る（ xlab = ””，ylab = ”” を同時に指定した場合と同じ．)．
tmag=1.2	プロットの別の注釈するテキストに関する主なタイトルのテキストの拡大率を指定する.

以下に例を示す.

```
> x <- rnorm(10)
> plot(x, main="Simple Time Series")
```

11.9　図の重ね合わせ

図を重ね合わせるには，関数 par() を用いるか，引数に add=T を入れる．

(注) plot() 以外の関数は add=T が効かない場合が多いので注意.

```
> plot(sin, -pi, pi, xlab="", ylab="", lty=2)   # sin(x) を描く
> par(new=T)                                    # 上書き指定
> plot(cos, -pi, pi, xlab="x", ylab="y")        # cos(x) を上書き
> plot(sin, -pi, pi, xlab="x", ylab="y", lty=2) # 新たに sin(x) を描く
> plot(cos, -pi, pi, add=T)                     # cos(x) を上書き
```

11.10　点の色を条件に応じて変える

点の色を条件に応じて変える場合は以下のように条件分岐すればよい.

```
> x <- runif(100)
> y <- runif(100)
> plot(x, y, col = ifelse(y>0.5, "red", "blue")) # y > 0.5 なら赤，その他は青
```

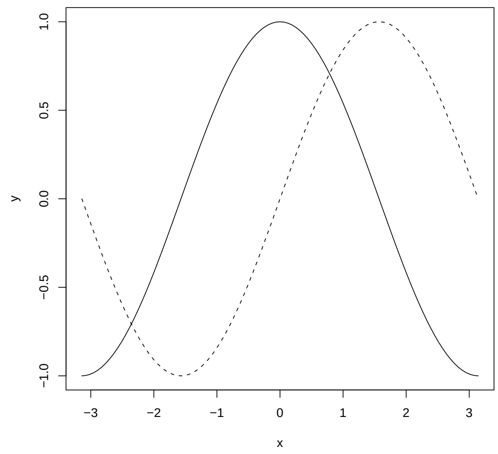


図 11.3　図の重ね合わせ

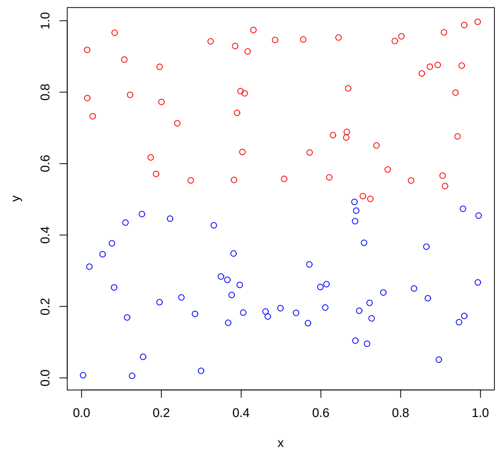


図 11.4　点の色を条件に応じて変える

11.11　図の消去

プロットした図を消去するには関数 frame() または関数 plot.new() を使う．グラフィックスパラメータ new が TRUE ならば図は消去されない.

```
> frame()
> plot.new()
```

第 12 章

高水準作図関数

以下に種々の高水準作図関数を紹介する.

12.1　散布図

12.1.1　plot()

plot() の詳しい説明は前節を御覧頂きたい．ここでは，plot() への引数の与え方による出力の違いを見てみる.

plot(x) ：　ベクトル x の要素が実数ならば，x は時系列データとみなされ，横軸を自然数，縦軸をデータ x 要素とする時系列プロットが描かれる.

- ベクトル x が時系列データならば，そのまま時系列プロットが描かれる.
- ベクトル x の要素が複素数ならば，横軸を実数，縦軸を虚部とするプロットが描かれる.
- x が 2 列の行列ならば，横軸を一列目，縦軸を 2 列目とするプロットが描かれる.
- x が 2 次元リストならば，その要素を横軸，縦軸としてプロットが描かれる．ただし names() を使ってどちらが x なのか y なのかラベルをつける必要がある.

```
> plot(rnorm(10))
> x <- list(1:5, 3:7);   names(x) <- c("X", "Y")
> plot(x)
```

plot(x, y) ：　データが入っているベクトル x，y やリスト x，y を点の座標として与えると散布図を描く.

```
> x <- rnorm(10);   y <- rnorm(10)
> plot(x,y)
```

plot(y ~ x) ：　以下の様に回帰式として入力することも出来る.

```
> x <- rnorm(10);   y <- rnorm(10)
> plot(y ~ x)
```

plot(f) ：　f は因子オブジェクトである．f の棒グラフを描く.

```
> f <- factor(c(rep("A",3), rep("B",5)))
> plot(f)
```

plot(f, y) ：　f は因子オブジェクト，y は数値ベクトルである．f の各水準に対する y の箱ひげ図を描くときに使う.

```
> x <- factor(c(rep("A",3), rep("B",5)));   y <- rnorm(8)
> plot(f, y)
```

plot(df)： df はデータフレームである．データフレーム中の変量のプロットを行う．

plot(~ expr)： expr は "+" で仕切られたオブジェクト名のリスト (例えば a+b+c) である．名前が与えられたオブジェクトの分布関数のプロットを行う．

```
> plot(~ group, data=sleep)
> plot(~ extra, data=sleep)
```

plot(y ~ expr)： y は任意のオブジェクト，expr は "+" で仕切られたオブジェクト名のリスト（例えば a+b+c ）である．expr に名前が与えられた全てのオブジェクトに対して y のプロットを行う．

12.1.2 dotchart()

関数 dotchart() でドットチャートを描く棒グラフの棒の代わりに点でプロットする[*1]．

```
dotchart(データ, groups=グループのベクトル,
         gdata=NULL, labels=ラベル, ...)
```

```
> dotchart(1:10, group=c(rep(1:2,5)),
+          labels=paste("sample", 1:10))
```

12.1.3 stripchart()

関数 stripchart() で因子レベル別に一次元散布図を描く

```
stripchart(データ ~ 因子ベクトル,
           vertical = F, group.names=,...)
```

```
> y <- rnorm(20); x <- factor(rep(1:2,10))
> stripchart(y ~ x)
```

12.1.4 sunflowerplot()

関数 sunflowerplot() でヒマワリ図を描く．これは 1 点の周りに複数データが対応する際に，点の周りに重なった分だけ花弁を描く．

```
sunflowerplot(x 軸データ, y 軸データ, digits=6,
              xlab = "", xlim = NULL, add = F, ...)
```

```
> x <- round(rnorm(50), d=1)
> y <- data.frame(x, x)
> sunflowerplot(y)
```

12.1.5 curve()

関数 curve() を用いれば，関数を直接指定してグラフを出力することも出来る．引数 n でポイントの数を指定することが出来る．

```
curve(関数, from=下限, to=上限, n = 点の数,
      add = FALSE, type = "l", xlim = NULL, ...)
```

```
> curve(sin(x^2)*exp(-x^2),-pi,pi)
```

12.1.6 matplot()

matplot() は行列を引数にとり，その各列についてプロットを行う．完成図は，座標設定や列ごとにマーカーの色や形を自動的に設定し，見た目に区別がつくように描かれる．また，上書き用に 関数 matpoints()，matlines() が用意

[*1] 似たような関数に rug() がある

```
plot(density(faithful$eruptions))
rug(faithful$eruptions, side=1)
```

されている．

```
matplot(行列（もしくは 2 つのベクトル）,
        type = "p", lty = 1:5, lwd = 1,
        pch = NULL, col = 1:6, ...)
```

```
> myfunc <- function(x,y) sin(x/20 * pi * y)
> sines  <- outer(1:20, 1:4, myfunc)
> matplot(sines, pch = 1:4, type = "o")
```

12.1.7　出力結果一覧

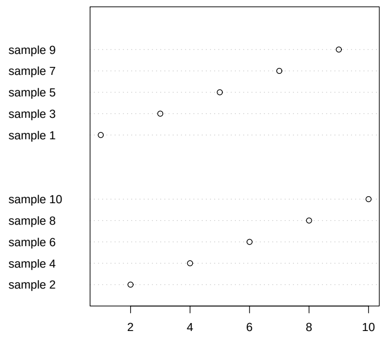


図 12.1　dotchart()

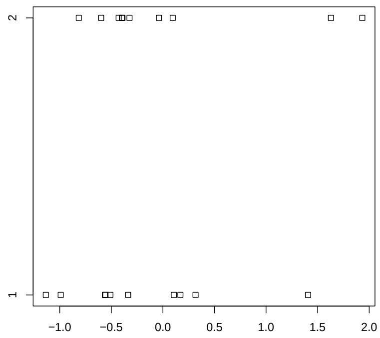


図 12.2　stripchart()

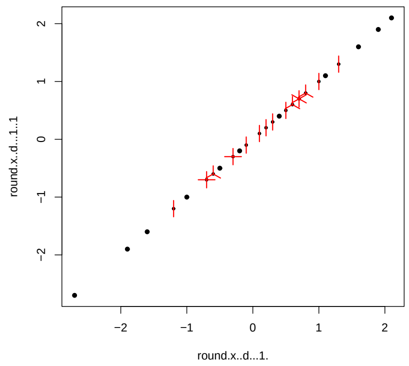


図 12.3　sunflowerplot()

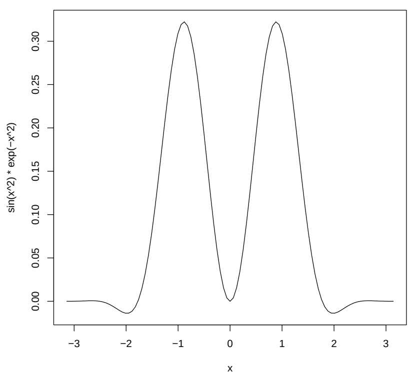


図 12.4　curve()

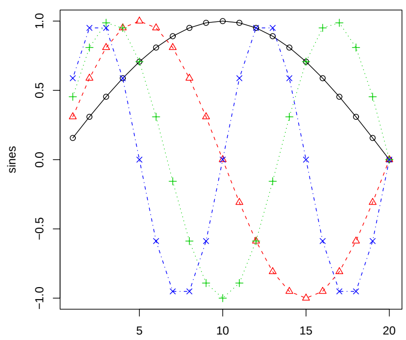


図 12.5　matplot()

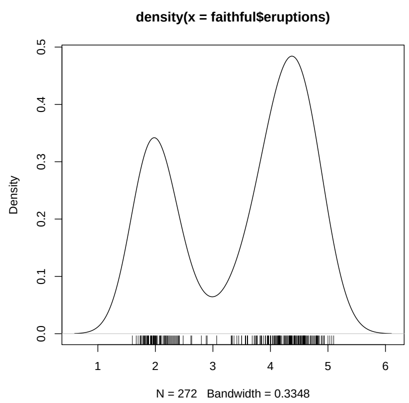


図 12.6　rug()

12.2　一次元データの表現

12.2.1　ヒストグラム：hist()

関数 hist() でヒストグラムを描くことが出来る．異なる区切り幅のヒストグラムを描くことも出来るが，主観的なバイアス (偏り) が入る危険があるのでお勧めは出来ない[*2]．

```
> x <- rnorm(50)                        # 一次元データ
> hist(x, breaks = seq(-3,3,1) )        # -3 から -3 まで 1 ずつの幅で描く
> hist(x, breaks = c(-3,-1,0,0.5,3) )   # 異なる区切り幅（出力例は省略）
```

[*2] 詳しくは『統計解析』の章で述べる．

12.2.2 棒グラフ：barplot()

関数 barplot() で棒グラフ描くことが出来る．データにベクトルを指定すると各要素の長さについて棒グラフが描かれ，データに行列を指定すると，一本（一列）の棒に各行の要素が層別で表示される．

```
barplot(ベクトルデータ, ...)
barplot(行列データ, ...)
```

```
> barplot(1:10)
> barplot(matrix(1:20, 5), col=rainbow(5))
```

関数 barplot() のオプションは以下の通り．ところで，barplot() は各棒の中心位置の x 座標を返すので，この値と lines などの低水準作図関数を使うことで，例えば信頼区間を描くことが出来る．

引数	機能
angle=45	棒を斜線で塗る際の線分の角度を指定する．
beside=F	（行列データで層別表示する場合）T にするとグループ毎に棒が並べて表示され，F にすると一本の棒に層別で表示される．
col=rainbow(10)	棒を塗りつぶす際の色を指定する．（ベクトルでも可）
density=30	棒を斜線で塗る際の線分の密度を指定する．
horiz=F	T にすると棒が水平に表示される．
legend = rownames(x)	凡例を描く．（例はデータ名が x の場合）
names=文字型ベクトル	各棒のラベルを文字型ベクトルで指定する．
space=1	各棒の間の間隔を指定する．
width=数値型ベクトル	各棒の相対的な幅を指定する．

12.2.3 円グラフ：pie()

関数 pie() で円グラフ描くことが出来る．

```
pie(データ, labels = names(x), radius = 0.8,
          density = NULL, angle = 45, col = NULL, ...)
```

```
> pie(1:10, r=1,
      col=rainbow(10))
```

関数 pie() のオプションは以下の通り．

引数	機能
angle=45	棒を斜線で塗る際の線分の角度を指定する．
border=NULL	NA にすると仕切り線が描かれなくなる．
col=rainbow(10)	棒を塗りつぶす際の色を指定する．（ベクトルでも可）
density=30	棒を斜線で塗る際の線分の密度を指定する．
main="Title"	タイトルを指定する．
radius=0.8	円の半径を −1〜1 の範囲で指定する．負の値にすると，データを表示する開始位置が左端からとなる．

12.2.4 箱ひげ図：boxplot()

複数のデータの分布の違いを比較する際は，関数 boxplot() で箱ひげ図を描く．

```
boxplot(x, range = 1.5, width = NULL,
        horizontal = FALSE, add = FALSE,...)
boxplot(x ~ 因子ベクトル)
```

```
> x <- rnorm(100)
> boxplot(x)
> boxplot(len ~ dose, data = ToothGrowth)
```

関数 boxplot() のオプションは以下の通り．

引数	機能
boxwex = 0.8	箱の幅を指定する．
horizontal=F	T にすると，箱が横向きに描かれる．
notch=F	箱に notch を入れるかどうかを指定する．
outwex = 0.5	outlier の幅を指定する．
range=1.5	「ひげ」が箱の端からどれだけ離れるかを指定する．range が正の値であるなら，「ひげ」は箱から四分位範囲の range 倍（デフォルトは 1.5 倍）である最も端にあるデータ点となる．例えば，上側の「ひげ」の場合は Q3 + 1.5×(Q3−Q1) となる．range が 0 ならば「ひげ」はデータ範囲の一番端の値になる．
staplewex = 0.5	ひげの幅を指定する．
width=数値ベクトル	箱の幅を指定する．

12.2.5　出力結果一覧

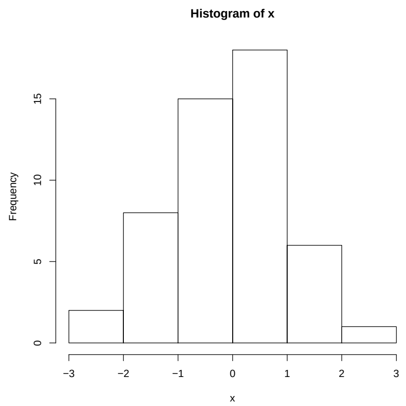


図 12.7　hist()

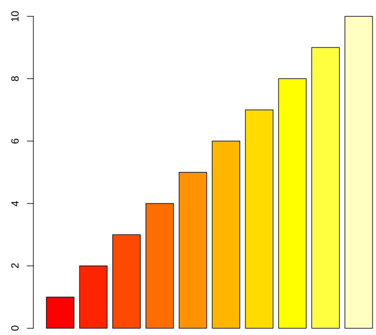


図 12.8　barplot()−1

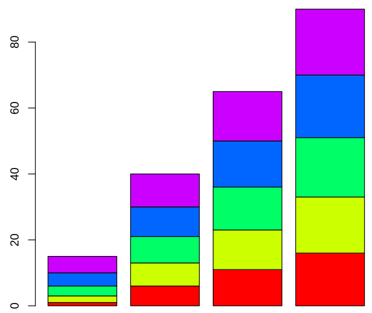


図 12.9　barplot()−2

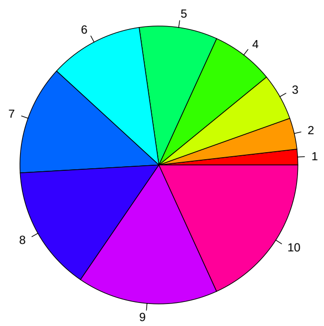


図 12.10　pie()

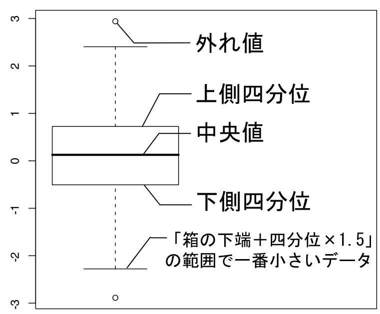


図 12.11　boxplot()−1

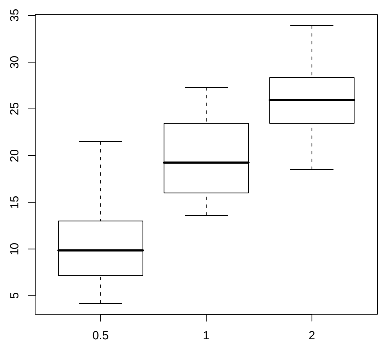


図 12.12　boxplot()−2

12.3　分割表データの図示

12.3.1　fourfoldplot()

関数 fourfoldplot() で 1 個以上の層について，2 変数間の関係を考慮に入れた 2×2 のグラフを生成する．

```
fourfoldplot(2*2 分割表データ)
```

```
> x <- matrix(1:4,2)
> fourfoldplot(x, col=1:2)
```

12.3.2　mosaicplot()

関数 mosaicplot() で 分割表データをモザイクプロットとして表示する．引数 dir=”h” (=”v”) で分割方向が変わる．

```
mosaicplot(分割表データ)
mosaicplot(モデル式, data=データ名)
```

```
> mosaicplot(Titanic, color = T)
> mosaicplot(~ Sex+Age+Survived, data = Titanic)
```

12.3.3　assocplot()

関数 assocplot() で分割表のデータについて，Cohen-Friendly の連関プロット（Association Plots）を行う．

```
assocplot(分割表データ)
```

```
> x <- margin.table(HairEyeColor, c(1,2))
> assocplot(x)
```

12.3.4　出力結果一覧

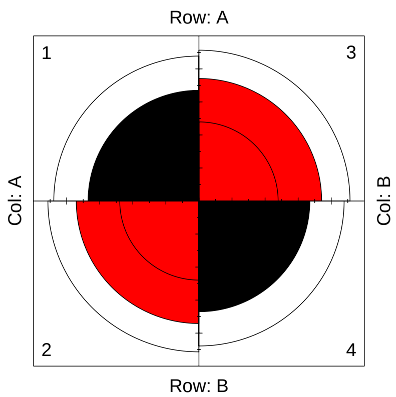


図 12.13　fourfoldplot()

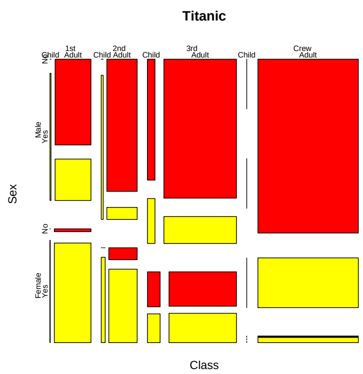


図 12.14　mosaicplot()

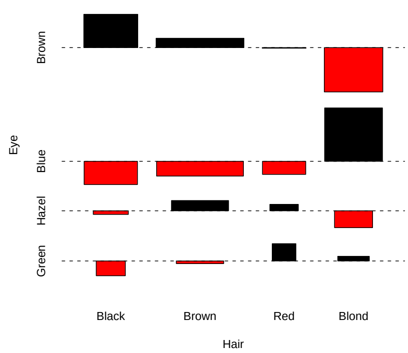


図 12.15　assocplot()

12.4　多変量データの図示

12.4.1　stars()

関数 stars() を用いることで，くもの巣プロットを描いて全体の傾向を見たり，星形図を描いてサンプルの分類を行うことが出来る．

```
> stars(mtcars[, 1:6], locations = c(0,0), radius = T,
+       key.loc=c(0,0.0), main="Motor Trend Cars")        # くもの巣プロット
> stars(mtcars[, 1:7], len = 0.8, key.loc = c(12, 1.5),
+       main = "Motor Trend Cars", draw.segments = TRUE)  # 星形図
```

12.4.2　symbols()

関数 symbols() で多変量データを図示する散布図を描くことが出来る．ただし，点や線の代わりに，円や星，箱ひげ図で散布図が描かれる．

```
> x <- 1:10;  y <- sort(10*runif(10));  z <- runif(10)
> symbols(x, y, thermometers=cbind(0.5,1,z), inches=0.5, fg=1:10)
```

温度計の記号以外にも様々な記号で表すことが出来る．

記号・機能	引数の指定方法
円	circles=数値ベクトル（半径）
正方形	squares=数値ベクトル（半径）
長方形	rectangles=数値ベクトル（半径）
星	stars=3 列以上の行列（星の中央からの線の長さ，0 以上 1 以下）
温度計	thermometers=c(幅，高さ，塗りつぶす高さの比率) または thermometers=c(幅，高さ，塗りつぶす比率 1，塗りつぶす比率 2)
箱ひげ図	boxplots=c(幅，高さ，下ひげの長さ 1，上ひげの長さ 2，中央線の位置)
シンボルの色	fg=数値ベクトル
塗りつぶす色	bg=数値ベクトル
単位	inches=F ならば単位は x 軸の単位が使われる．inches=T ならば最大のシンボルが高さ 1 インチであるようにシンボルが指定される．inches=数値ならば最大のシンボルが（インチ単位で）指定した数値の高さになるように調節される．

12.4.3　pairs()

関数 pairs(行列) で各列同士の組合せ全てについて散布図を描く．関数 pairs(モデル式) で，細かい条件を指定した上で，各列同士の組合せ全てについて散布図を描く．

```
> pairs(iris[1:4])                                          # 行列で指定
>
> pairs(~ Fertility + Education + Catholic, data = swiss,
+       subset = Education < 20, main = "Education < 20")  # モデル式で指定
```

12.4.4　coplot()

関数 coplot() で共変量プロットを描く．この関数は数値ベクトル x，y，z (z は因子オブジェクトでも可) を引数としてとり，z の与えられた値における y に対する x の散布図を複数生成する．

coplot(y ˜ x | z) ：　z が因子ならば，z の水準ごとに y が x に対してプロットされる．z が数値ならば，いくつかの『条件を与える区間』に分割され，その区間に含まれる z の値に対して y が x に対してプロットされる．

coplot(y ˜ x | $z_1 * z_2$) ：　2 変量 z_1，z_2 に条件付けされた，x に対する y の散布図を生成する．

```
> x <- 1:10;  y <- rnorm(10);  z <- x*y
> coplot(y ~ x | z)
```

12.4.5　出力結果一覧

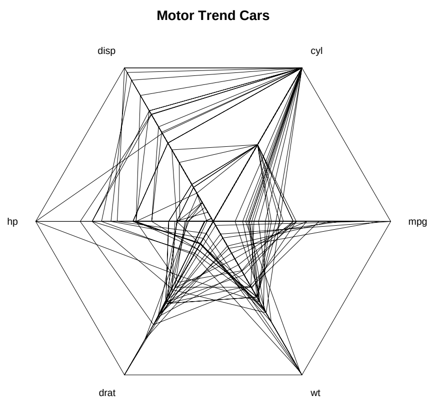


図 12.16　stars()−1

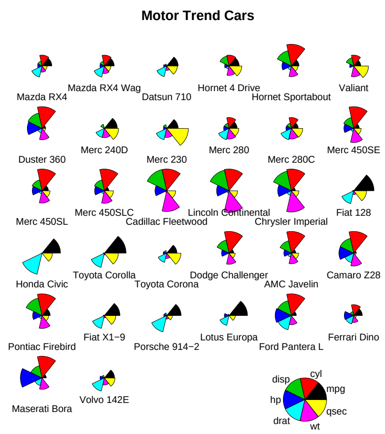


図 12.17　stars()−2

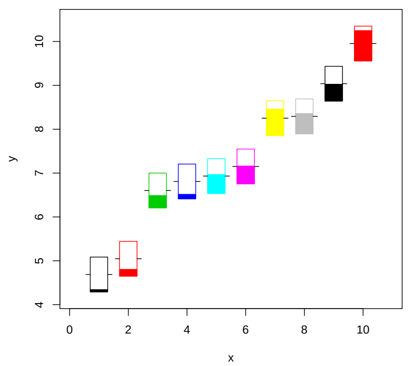


図 12.18　symbols()

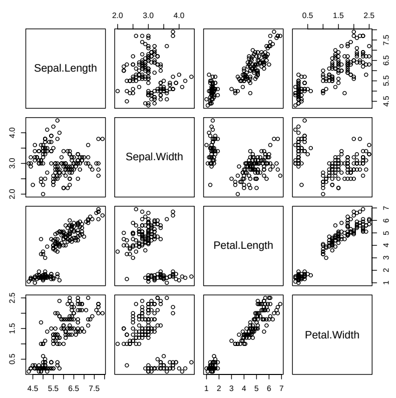


図 12.19　pairs()−1

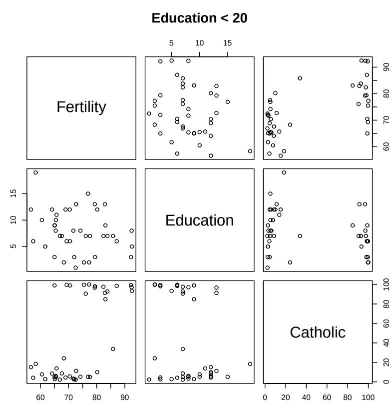


図 12.20　pairs()−2

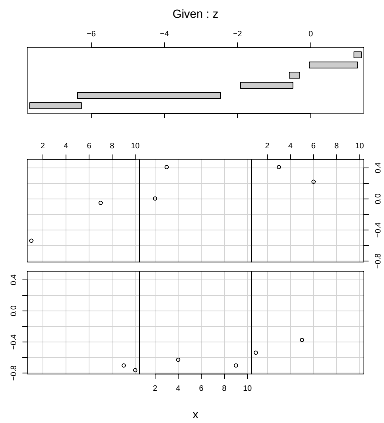


図 12.21　coplot()

12.5　3 次元データの図示

12.5.1　image()

関数 image() で 3 次元データをカラーイメージで図示する．

```
> x <- 10*(1:nrow(volcano));   y <- 10*(1:ncol(volcano))
> image(x, y, volcano, col = terrain.colors(100), axes = FALSE)
```

12.5.2　persp()

関数 persp() で 3 次元立体図を描く．

```
> x <- seq(-10, 10, length= 50);   y <- x
> f <- function(x, y) { r <- sqrt(x^2+y^2); 10 * sin(r)/r }
> z <- outer(x, y, f);
> persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = rainbow(50), border=NA)
```

12.5.3　contour()

関数 contour() で 3 次元データを等高線図で図示する．関数 filled.contour() も同様の関数である．

```
> x  <- -6:16
> contour(outer(x, x), method = "edge", vfont = c("sans serif", "plain"))
```

12.5.4　scatterplot3d()

パッケージ scatterplot3d には 3 次元的に表示する散布図を描く関数 scatterplot3d() が入っている．まず，パッケージ scatterplot3d を呼び出す必要がある．

```
> library(scatterplot3d)
> z <- seq(-10, 10, 0.01);  x <- cos(z);  y <- sin(z)
> scatterplot3d(x, y, z, highlight.3d=TRUE, col.axis="blue",
+               col.grid="lightblue", main="Plot-1", pch=20)  # 線プロット
>
> temp <- seq(-pi, 0, length = 50)
> x <- c(rep(1, 50) %*% t(cos(temp)))
> y <- c(cos(temp) %*% t(sin(temp)))
> z <- c(sin(temp) %*% t(sin(temp)))
> scatterplot3d(x, y, z, highlight.3d=TRUE, col.axis="blue",
+               col.grid="lightblue", main="Plot-2", pch=20)  # 点プロット
```

12.5.5　【参考】パッケージ lattice

lattice には S 用に開発されたグラフィックスパラダイム Trellis の R への移植関数が入っている．例えば，三次元散布図は，関数 cloud() や関数 wireframe() で描くことが出来る．

```
> library(lattice)
> data(volcano)
> wireframe(volcano, shade = TRUE, aspect = c(61/87, 0.4),
+           light.source = c(10,0,10))
```

lattice には以下のような関数が用意されている．

種類	関数名	機能
一次元データ用	barchart()	棒グラフ
	bwplot()	箱ひげ図
	densityplot()	密度関数の推定
	dotplot()	ドットプロット
	histogram()	ヒストグラム
	qqmath()	分布関数のプロット
	stripplot()	1 次元散布図
モデルへの当てはめ	rfs()	残差プロット

種類	関数名	機能
二次元データ用	qq()	2 次元 q-q プロット
	xyplot()	trellis 版 plot()
等高線プロット	levelplot()	等高線プロット
	contourplot()	等高線プロット
三次元データ用	cloud()	三次元散布図 (点)
	wireframe()	三次元散布図 (面)
高次元データ用	splom()	行列の散布図
	parallel()	パラレルプロット

12.5.6　出力結果一覧

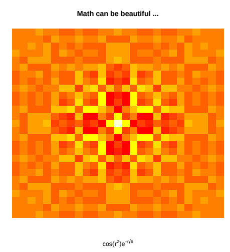


図 12.22　image()

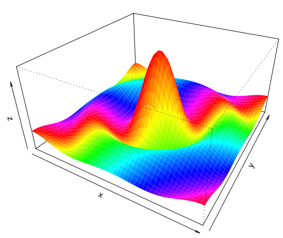


図 12.23　persp()

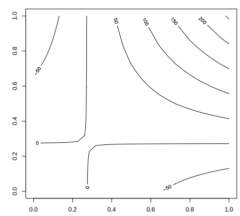


図 12.24　contour()

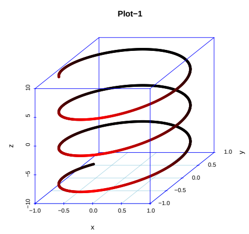


図 12.25　scatterplot3d()−1

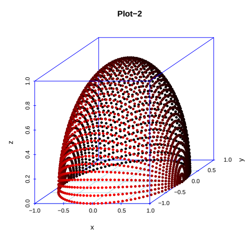


図 12.26　scatterplot3d()−2

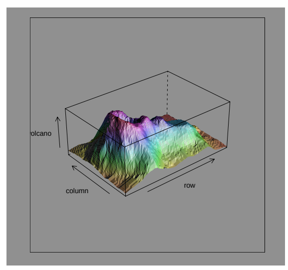


図 12.27　lattice・wireframe()

12.5.7　【一言】パッケージ grid

パッケージ grid を用いることで，グリッド・グラフィックスの世界で作図を行うことが出来る．この利点は，R の graphics パッケージでは自由度に制限があったカスタマイズが自由に出来る点である．graphics パッケージ中の高水準作図関数や低水準作図関数では十分な作図が得られない時に力を発揮し，例えば lattice パッケージ中の作図関数にはグリッド・グラフィックス中の関数が頻繁に使われている[*3]．

[*3] 詳しくは，グリッド・グラフィックスパッケージの作者 Paul Murrell 先生のホームページ（*http : //www.stat.auckland.ac.nz/ paul/*）にある PDF ドキュメントを参照のこと．

第 13 章

低水準作図関数

高水準関数で描いた図に文字や図形を描き加える場合は低水準作図関数を用いる．低水準作図関数はグラフィックスパラメータを引数として受け入れることも出来る．

13.1　低水準作図関数一覧

まず，低水準作図関数の一覧表を挙げる．

種類	関数	機能
点	points(x, y) , points(c(x, y))	各点の x 座標と y 座標を指定することで点列を描く (規定では points() に対して，関数の引数 type に ”p” を与える) ．マーカーの形式はグラフィックスパラメータ pch によって指定する．また，points(approx(x, y)) でデータの線形補間が行える．
直線	lines(x, y) , lines(c(x, y))	各点の x 座標と y 座標を指定することで，それぞれの点を通る直線・曲線を描く．点列を描く (規定では lines() に対して，関数の引数 type に ”l” を与える) ．
直線	abline(a, b), abline(c(a, b))	a ，b は切片と傾きを表し，直線 y = a + bx を描く．
直線	abline(h = y)	ベクトルで与えられた y 座標の水平線を引く．
直線	abline(v = x)	ベクトルで与えられた x 座標の垂直線を引く．
直線	abline(result)	result は関数 lm() で直線回帰を行った結果が入ったオブジェクトで，回帰直線が描かれる．
格子	grid(a, b)	a × b 本の格子を描く．引数として col, lty, lwd が指定できる
線分	segments(x0, y0, x1, y1)	始点の座標 (x0，y0) と，終点の座標 (x1 ，y1) を通る線分を描く．
矢印	arrows(x0, y0, x1, y1)	始点の座標 (x0，y0) と，終点の座標 (x1 ，y1) を通る矢印を描く．
矩形	rect(x0, y0, x1, y1)	始点の座標 (x0，y0) と，終点の座標 (x1 ，y1) を通る長方形を描く．引数として col, border, density が指定できる
文字	text(x, y, labels)	座標 (x，y) にラベルが描かれる．座標値と文字列をベクトルで指定することも出来る．text の引数に srt = − (回転角) を入れることで，プロットの x 軸ラベルを 45 度回転させることが出来る．
文字	mtext(text, side = 3, line = 0, at = NA)	書き込む文字列を引数 text で指定し，side に文字列を書き込む余白位置を表す番号 (1：下，2：左，3：上，4：右) を指定する．ここで line には図形領域から何行離すかを指定し，at には文字列を書き込む座標を指定することも出来る．

種類	関数	機能
枠	box()	軸や目盛を描かず枠だけ描く．例えば box(lty='1373', col = 'red') として色や線の太さを指定することも出来る．(この場合は図の枠に赤い模様が付く)．
題名	title(main, sub)	引数 main に上部余白に描かれるメインタイトルを，引数 sub に下部余白に描かれるサブタイトルを指定して，図の上下の余白部分にタイトルを追加する．省略すると，それぞれメインタイトルとサブタイトルが描かれなくなる．また，引数 tmag でテキストの拡大率を指定することが出来る．
軸	axis(side=4, labels=F)	座標を描く (1：下，2：左，3：上，4：右)．引数 labels に FALSE を指定すると目盛のラベルは描かれなくなる．また，引数 at=1:10 で目盛のラベルを指定することが出来る．
軸	axis(side=1, pos=0)	引数 pos で軸を描く位置を指定することが出来る．上 (side=3) か下 (side=1) に軸を描く場合は y 座標を，右 (side=4) か 左 (side=2) に軸を描く場合は x 座標を pos に与えればよい．例えば pos = 0 とすれば原点を通る座標軸を描くことが出来る．
凡例	legend(x, y, legend)	座標 (x，y) に凡例を追加する．引数 legend に文字ベクトルを指定することにより複数行に渡る凡例を，引数 ncol に 2 以上の数値を指定することで複数列に渡る凡例を作ることが出来る．
凡例	legend(locator(1), legend=文字)	関数 locator() により対話的に凡例の位置を決定する．
凡例	legend(..., fill=v , col=v)	箱を塗りつぶす色，点や線を描く色を指定する．
凡例	legend(..., lty=v , lwd=v , pch=v)	線の種類と線の幅，プロット用の文字 (文字ベクトル) を指定する．
図形	polygon(x,y)	(x，y) に多角形の頂点の座標ベクトル（または x，y 成分を持つリストや行列）を指定して多角形を描いて中を塗りつぶす．要素に NA があると，多角形の生成は終了する．
図形	polygon(x, y, density=c(10, 20), angle=c(−45, 45))	引数 density により，ビット/インチで太さを指定したラインを使って多角形の内部に影を入れることが出来る．このとき，引数 angle で角度 (左回りに) を与えて線の傾斜を指定することが出来る．引数は border や col , lty や xpd を指定することが出来る．

13.2　使用例

(例 1) まず，散布図を描いた後，まず関数 axis() で 両軸を描き，次に関数 box() で枠を描く．さらに関数 legend() で凡例を描いた後，最後に関数 polygon() で多角形を描いている．

```
> plot(rnorm(50), rnorm(50), xlim=c(-3,6), ylim=c(-3,3), axes = F, ann=F)
> axis(1, pos = 0, at = -3:6, adj = 0, col = 2)        # 赤で X 軸を描く
> axis(2, pos = 0, at = -3:6, adj = 1, las = 2)        # 黒で Y 軸を描く
> box()
> legend(2, 3, paste("sin(",6:9,"x)"), col=6:9,
+        pch=3, ncol=2, cex=1.1, pt.bg="pink")         # (5,9) に凡例を描画
> polygon(3:6, c(-2,-1,-2,-1), density=c(10, 20), angle=c(-45, 45))
> arrows(5, -1, 4, 2, col="blue")                      # (5,-1) から (4,2) に矢印を描く
```

(例 2) 関数 polygon() を用いることで，グラフの一部に影を付けることが出来る[*1]．まず，標準正規分布の密度関数のグラフを $-4 \leq x \leq 4$ の範囲でプロットする．そのうち $2 \leq x \leq 4$ の範囲に灰色の影を付けるには，以下の様に多数の多角形に分割して関数 polygon() で塗りつぶしをすればよい．具体的には xvals，rep(0,10) の組合せで x 軸上の辺を結び，rev(xvals)，rev(dvals) の組合せでグラフに沿った辺を結べばよい．

```
> plot(dnorm, -4, 4)
> xvals <- seq(2, 4, length=10)        # 領域を x 軸方向に 10 個の多角形（台形）に等分割
> dvals <- dnorm(xvals)                # 対応するグラフの高さ
> polygon(c(xvals,rev(xvals)),
+ c(rep(0,10),rev(dvals)),col="gray") # 塗りつぶす
> title("Title")                       # タイトルを描く
> mtext("sub-title", side=4)           # 文字を描く
```

以下に (1) と (2) の出力結果を載せる．

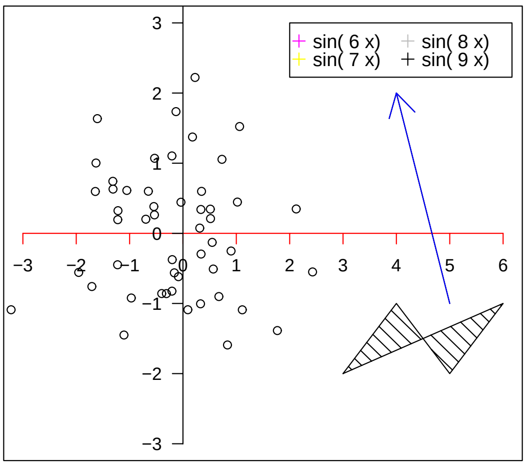


図 13.1　例 (1)

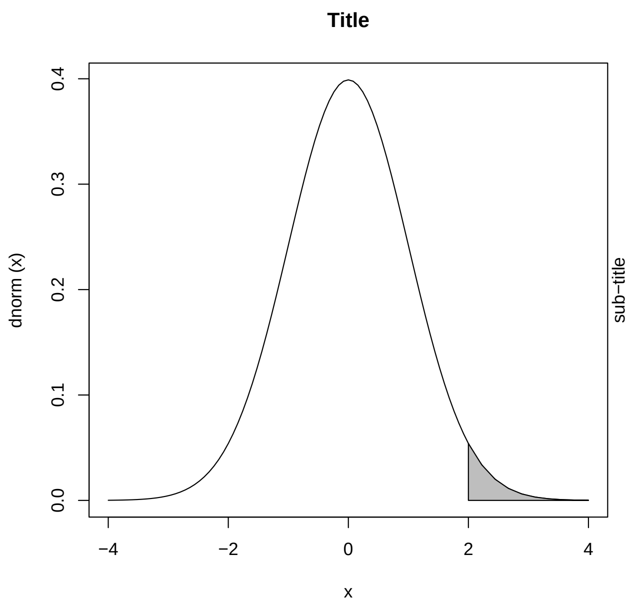


図 13.2　例 (2)

13.3　数式の描画

プロットに数学記号や式を使いたい場合で text()，mtext()，axis()，title() のいずれかを使う場合は，文字列の代わりに expression() を指定すればよい．例えば以下は二項分布の確率関数の公式を書く．

```
> text(x,y,expression(paste(bggroup("(",atop(n,x),")"),p^x, q^{n-x})))
```

一覧はオンラインマニュアルで御覧頂ける．

```
> ?plotmath
> demo(plotmath)
> example(plotmath)
```

以下に書式と出力例を挙げる．

[*1] RjpWiki の記事より引用した．

Arithmetic Operators	
x + y	$x+y$
x - y	$x-y$
x * y	xy
x/y	x/y
x %+-% y	$x \pm y$
x%/%y	$x \div y$
x %*% y	$x \times y$
-x	$-x$
+x	$+x$
Sub/Superscripts	
x[i]	x_i
x^2	x^2
Juxtaposition	
x * y	xy
paste(x, y, z)	xyz
Lists	
list(x, y, z)	x, y, z

Radicals	
sqrt(x)	$\sqrt{x}$
sqrt(x, y)	$\sqrt[y]{x}$
Relations	
x == y	$x = y$
x != y	$x \neq y$
x < y	$x < y$
x <= y	$x \leq y$
x > y	$x > y$
x >= y	$x \geq y$
x %~~% y	$x \approx y$
x %=~% y	$x \cong y$
x %==% y	$x \equiv y$
x %prop% y	$x \propto y$
Typeface	
plain(x)	x
italic(x)	*x*
bold(x)	**x**
bolditalic(x)	***x***

Style	
displaystyle(x)	$\displaystyle x$
textstyle(x)	$\textstyle x$
scriptstyle(x)	$\scriptstyle x$
scriptscriptstyle(x)	$\scriptscriptstyle x$
Spacing	
x ~ ~y	$x\ \ y$

x + phantom(0) + y	$x + + y$
x + over(1, phantom(0))	$x + \frac{1}{}$
Fractions	
frac(x, y)	$\frac{x}{y}$
over(x, y)	$\frac{x}{y}$
atop(x, y)	${x \atop y}$

Big Operators	
sum(x[i], i = 1, n)	$\sum_{1}^{n} x_i$
prod(plain(P)(X == x), x)	$\prod_{x} P(X = x)$
integral(f(x) * dx, a, b)	$\int_a^b f(x)dx$
union(A[i], i == 1, n)	$\bigcup_{i=1}^{n} A_i$
intersect(A[i], i == 1, n)	$\bigcap_{i=1}^{n} A_i$
lim(f(x), x %->% 0)	$\lim_{x \rightarrow 0} f(x)$
min(g(x), x >= 0)	$\min_{x \geq 0} g(x)$
inf(S)	$\inf S$
sup(S)	$\sup S$

Grouping	
(x + y) * z	$(x+y)z$
x^y + z	$x^{y}+z$
x^(y + z)	$x^{(y+z)}$
x^{y + z}	x^{y+z}
group("(", list(a, b), "]")	$(a, b]$
bgroup("(", atop(x, y), ")")	$\left(\genfrac{}{}{0pt}{}{x}{y}\right)$
group(lceil, x, rceil)	$\lceil x \rceil$
group(lfloor, x, rfloor)	$\lfloor x \rfloor$
group("|", x, "|")	$\lvert x \rvert$

Ellipsis	
list(x[1], ..., x[n])	$x_1, \ldots, x_n$
x[1] + ... + x[n]	$x_1 + \cdots + x_n$
ist(x[1], cdots, x[n])	$x_1, \cdots, x_n$
x[1] + ldots + x[n]	$x_1 + \ldots + x_n$
Set Relations	
x %subset% y	$x \subset y$
x %subseteq% y	$x \subseteq y$
x %supset% y	$x \supset y$
x %supseteq% y	$x \supseteq y$
x %notsubset% y	$x \not\subset y$
x %in% y	$x \in y$
x %notin% y	$x \notin y$
Accents	
hat(x)	$\hat{x}$
tilde(x)	$\tilde{x}$
ring(x)	$\mathring{x}$
bar(xy)	$\overline{xy}$
widehat(xy)	$\widehat{xy}$
widetilde(xy)	$\widetilde{xy}$

Arrows	
x %<->% y	$x \leftrightarrow y$
x %->% y	$x \rightarrow y$
x %<-% y	$x \leftarrow y$
x %up% y	$x \uparrow y$
x %down% y	$x \downarrow y$
x %<=>% y	$x \Leftrightarrow y$
x %=>% y	$x \Rightarrow y$
x %<=% y	$x \Leftarrow y$
x %dblup% y	$x \Uparrow y$
x %dbldown% y	$x \Downarrow y$
Symbolic Names	
Alpha - Omega	$\mathrm{A} - \Omega$
alpha - omega	$\alpha - \omega$
infinity	∞
32 * degree	$32°$
60 * minute	$60'$
30 * second	$30''$

13.4　【戯言】eps ファイルの編集方法

R で出力した eps ファイルを編集する場合で，AD*BE ソフトなどの有料ソフトを使わない方法を紹介する．

13.4.1　Tgif を用いる方法

Tgif で eps ファイルを編集する場合は pstoedit ：*http://www.pstoedit.net/pstoedit* で eps 形式から obj 形式に変換してから編集する．ただし，Windows の場合は Tgif が最初から入っていないので，次の方法をお勧めする．

13.4.2　OpenOffice.org の draw を使う方法

セットアップの手順は以下の通り．

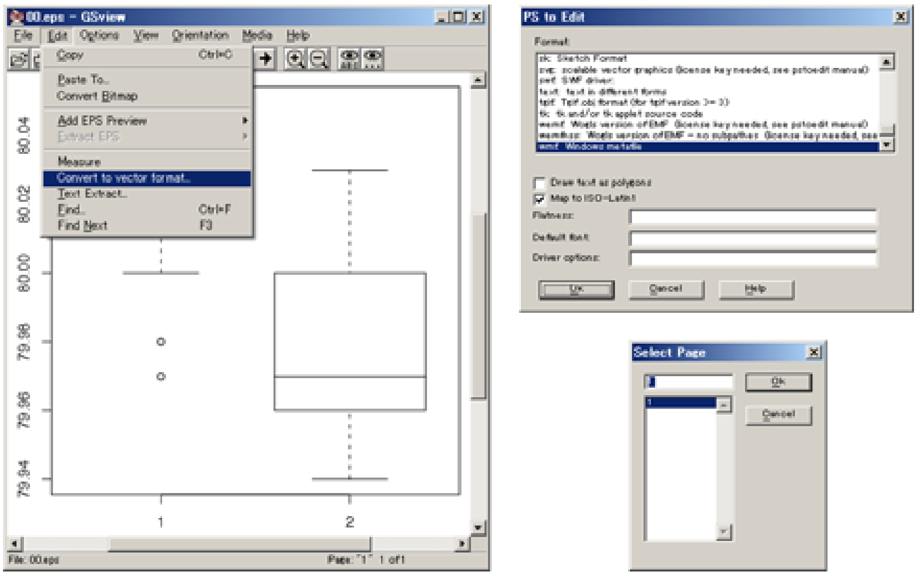
(1) Ghost Script (7.04 以降?) と GSView の設定：*http://forum.nifty.com/fdtp/install/win/gs800.htm* の設定を済ます．

(2) OpenOffice.org ：*http://ja.openoffice.org/marketing/news/110.html* の
OpenOffice.org の入り口 ：*http://oooug.jp/start/*
に従って OpenOffice.org1.1 (04/03/08 現在の日本語最新版) をダウンロードしてインストールする．

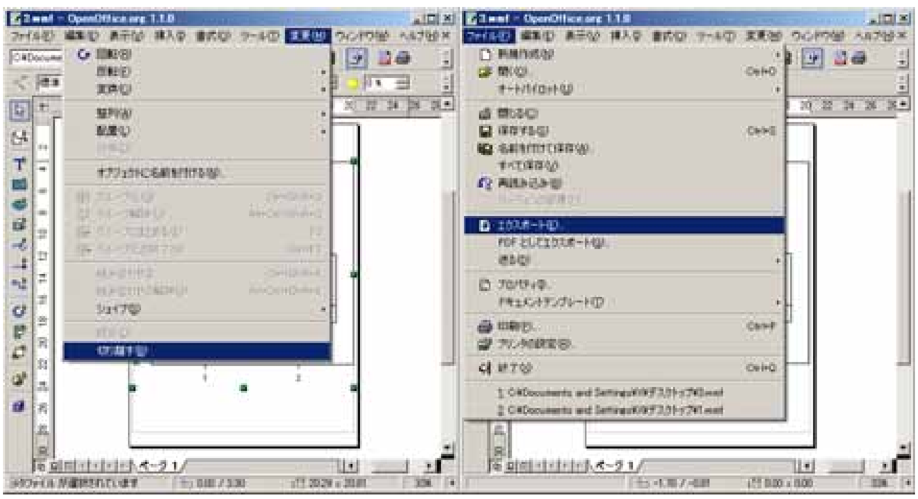
(3) pstoedit ：*http://www.pstoedit.net/pstoedit* を GSview フォルダの下に pstoedit フォルダを作ってそこに (GS 8.00 の標準では C:/Program Files/Ghostgum/pstoedit に) インストールする．

実際の動作手順は以下の通り．動作確認は Windows XP で行った．

(1) GSview のメニューから [Edit] → [Convert to vector format...] (左図) で WMF や EMF に変換 (右上・右下図で OK をクリック) する．(R のグラフを編集する場合は win.metafile() を使って emf 形式で出力する)



(2) OpenOffice.org の draw に読み込んで，メニューの [変更] → [切り離す] で切り離してから編集する．



(3) ページサイズや余白を変えてから eps 形式で export する．以上．

第 14 章

グラフィックスパラメータ

高水準作図関数や低水準作図関数で作図する場合，作図関数固有のパラメータ以外にもグラフィックスパラメータと呼ばれるパラメータを指定することが出来る．これにより作図結果の微妙なカスタマイズを行うことが出来，自分好みの出力結果を得ることが出来る．

14.1　グラフィックスパラメータ事始

グラフィックスパラメータを設定する方法は，作図関数の引数にパラメータを与える方法と，関数 par() を使って設定する方法の 2 通りがある．前者は一時的にパラメータ値が変更され，後者は永続的にパラメータ値が変更される．重要なことは，グラフィックスパラメータの全てがこの 2 通りの方法で変更出来るわけではなく，一部のグラフィックスパラメータは関数 par() を使ってしかパラメータ値を変更することが出来ない点を理解することである[*1]．例えば，色に関するグラフィックスパラメータ col を赤に設定する方法を挙げる．

```
> par(col="red")              # 永続的に（これ以後は）赤色でプロットする
> plot(1:10, col="red")       # 一時的に（このプロットのみ）赤色でプロットする
```

14.2　グラフィックスパラメータの永続的変更

まず，関数 par() を使わなければ変更できないグラフィックスパラメータを紹介する．この種のパラメータは領域，余白，座標系などを設定する機能であるものがほとんどであり，一旦変更すると，改めて変更するまでは元に戻らない．

引数	機能
par()	全てのグラフィックスパラメータの現在値がリストとして得られる．
par(”文字列”)	グラフィックスパラメータ名を文字列で指定すると，そのグラフィックスパラメータの現在値が得られる．グラフィックスパラメータの値を変更する際は，まずこの命令でパラメータ値を確認する．
par(”文字列”, ”文字列”) par(c(”文字列”, ”文字列”))	グラフィックスパラメータを複数指定することも出来る．
par(col=2)	「パラメータ名 = 値」の形で設定することでグラフィックスパラメータの値を変更する．
par(col=2, lty=3) par(list(col=2, lty=3))	複数のグラフィックスパラメータを一度に変更することも出来る．

[*1] 一部のグラフィックスパラメータには読み取り専用・変更不可なものがあるが，ここでは詳しく扱わないことにする．例えば，par(”cin”)，par(”cra”)，par(csi)，par(”cxy”)，par(”din”) でそれぞれ「インチ単位の既定文字サイズ（幅と高さ）」「ピクセル単位の既定文字サイズ（幅と高さ）」「インチ単位で与えた既定サイズの文字の高さ」「既定文字のサイズ（幅と高さ：par(”cin”)/par(”pin”))」「インチ単位のデバイスの大きさ（幅と高さ）」を確認することが出来るが，これらの値を明示的に変更することはできない．

14.2.1　関数 par() の使い方

par() の使い方を以下に挙げる[*2].

```
> par(new=T)          # new を TRUE に設定すると，現在の作図に次の作図を上書きする
> par(ask=TRUE)       # 作図する前に『作図してよいですか？』と質問させるようにする
> help(par)           # 全てのグラフィックスパラメータのヘルプが表示される
```

14.2.2　グラフィックスパラメータ値の一時退避と復帰

関数 par() を使わなければ変更できないグラフィックスパラメータの値を変更する場合は，適当な変数に現在のパラメータの値を保存しておくのが得策である．R では便利なことに，グラフィックスパラメータの値を 1 コマンドで一切合財保存することが出来る．

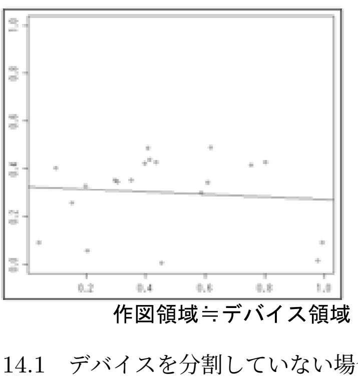
```
> oldpar <- par(no.readonly = TRUE)      # 現在のグラフィックスパラメータ値を退避する
> oldpar <- par(col=1, lty=2)            # 一部だけ保存する
> par(oldpar)                            # 作業前のグラフィックスパラメータ値に戻す
```

関数定義内で，関数 par() を使ってグラフィックスパラメータを変更する場合も，元のグラフィックスパラメータの値を一切合財保存した上で作業をし，関数の最後に元に戻すことをお勧めする．

```
> myfunc <- function () {
+   oldpar <- par(no.readonly = TRUE)    # 現在のグラフィックスパラメータの値を退避
+   on.exit(par(oldpar))                 # 関数がエラー中断してもパラメータを復帰
+   par(col=2);  plot(1:10)              # 色を赤に変更してからプロットを行う
+ }                                      # 関数を抜けた後にパラメータ値は元に戻る
```

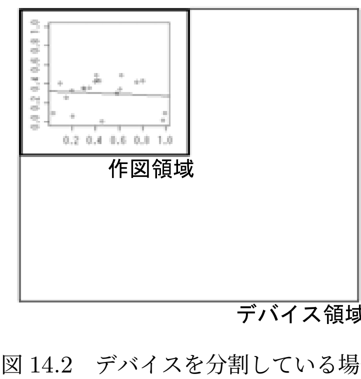
14.2.3　作図領域・余白・座標系

関数 par() を使わなければ変更できないグラフィックスパラメータは，大抵がプロット領域（関数 plot() で散布図を描いた場合：四角の枠とその中に描かれた点や線などがある領域）と余白（プロット領域の周囲，すなわち枠の外側の目盛や目盛のラベル，軸のラベルやタイトルなどが描かれる部分），作図領域（プロット領域 + 余白）及びデバイス領域に関するパラメータとなっている．以下に概念図を示す．



作図領域≒デバイス領域

図 14.1　デバイスを分割していない場合

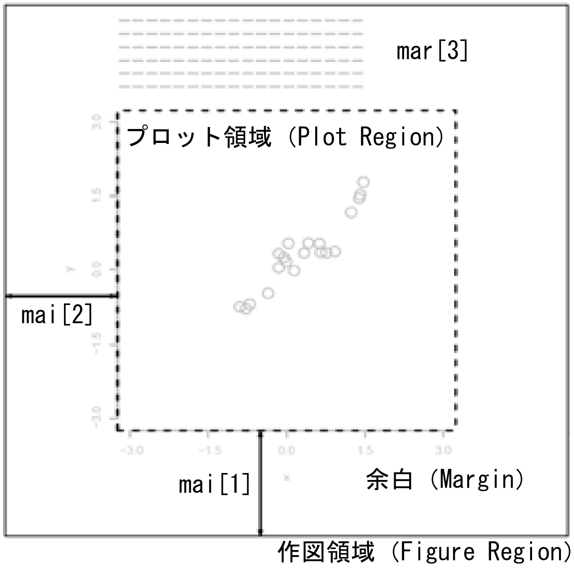


デバイス領域

図 14.2　デバイスを分割している場合

[*2] パラメータ new を TRUE にしておくと次に描く図は前の図に上書きして描かれ，パラメータ ask を TRUE にしておくと help の example を閲覧する際に作図例が一瞬で流れてしまうのを防ぐことが出来る．

デバイス領域とは，グラフを描いたときに表示されるウインドウの領域のこと[*3]で，普段はデバイス領域と作図領域はほぼ一致しているが，デバイスを分割している場合に違いが出てくる．次に，作図領域とプロット領域の違いを示す．



14.2.4　作図範囲に関するパラメータ

xpd = F に FALSE を指定するとプロット領域内に収まる部分だけ図が描かれ，TRUE を指定すると作図領域内に収まる部分だけ図が描かれる．NA を指定するとデバイス領域全体に図が描かれる．以下に例を示す．

```
> library(rpart)
> result <- rpart(Species ~ ., data=iris)   # 直前で par(xpd=F) 又は par(xpd=NA) を実行する
> plot(result)
> text(result)
```

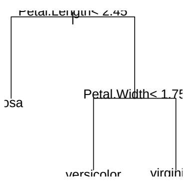


図 14.3　par(xpd=F) としている場合

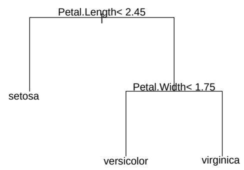


図 14.4　par(xpd=NA) としている場合

14.2.5　プロット領域（plot region）の大きさや位置を指定するパラメータ

引数	機能
pin = c(6,6)	プロット領域の幅と高さをインチ単位で指定する．
plt = c(0.1, 0.9, 0.1, 0.9)	プロット領域の x，y 軸方向の両端の位置の比率を指定する．c(0, 1, 0, 1) と変更すれば余白が全くない出力が得られる．
ps = 12	文字とシンボルの大きさを整数値で指定する．
pty = ”m”	”s” は正方形のプロット領域，”m” は最大のプロット領域を生成する．”m” を設定した場合，プロット領域でも余白でもない空白部分が出来る場合もある．

[*3] デバイスが ps, eps ならば，デバイス領域は用紙の大きさ（用紙の外周）となる．

ちなみに，par(”usr”) でプロット領域を知ることが出来る．

```
> par("usr")
```

14.2.6　作図領域（figure region）の大きさや位置を指定するパラメータ

引数	機能
fig = c(0, 1, 0, 1)	作図領域の x，y 軸方向の両端の位置の比率を指定する．初期値は c(0, 1, 0, 1) で，これは作図領域をフルに使用していることを示しており，c(0, 0.5, 0.5, 1) と変更すれば左上の方に作図されることになる．この値を設定した後，現在のプロットにグラフを描き加えるときは明示的に par(new=T) とする必要がある（この点は S と異なる）．
fin = c(5,5)	作図領域の幅と高さを指定する．例えば par(”fin”) で調べた結果が [6.968749 6.958332] であった場合に c(5, 5) を指定すれば，描画結果は少し小さめの出力となる．この値を設定した後，現在のプロットにグラフを描き加えるときは明示的に par(new=T) とする必要がある（この点は S と異なる）

14.2.7　余白に関するパラメータ

余白をあまり小さくしてしてしまうと軸のラベルなどが描けなくなってしまうので注意.

引数	機能
mai = c(0.85, 0.68, 0.68, 0.35)	底辺，左側，上側，右側の順に余白の大きさをインチ単位で指定する．mar で余白サイズを変えると plt，pin の値が変更される．
mex = 1	プロットの余白の座標を指定するのに使われる文字サイズの拡大率を指定する．このパラメータを変更すると，余白の大きさもそれに応じて変化する．複数図表を使う場合は，このパラメータ値が自動的に 0.5 に変更されることがあるので注意が要る．
mar = c(5, 4, 4, 2)	底辺，左側，上側，右側の順に余白の大きさを行の高さ（mex）で指定する．デフォルトでは高水準作図関数は軸のラベルを 3 行分離して描くので，底辺と左部には最低でも 4 行分 (4 mex 分) の余白が必要となる．mai で余白サイズを変えると plt，pin の値が変更される．
omi = c(0, 0, 0.8, 0)	底辺, 左側, 上側, 右側の順に外周の大きさをインチ単位で表した (outer margins in inches) もので指定する．
oma = c(2, 0, 3, 0)	外周の底辺, 左側, 上側, 右側の順に外側余白 (outer margin) を行の高さ (mex) で指定する．
omd = c(0, 1, 0, 1)	ウインドウ全体を 0 から 1 の範囲として，外周を除いた複数図表の両端の位置 (x_1, x_2, y_1, y_2) を指定する．この場合は外周は存在しないことになる．

例えば右側に y 軸座標を描く場合，グラフを描く前に mar の値を設定して，右側の余白を余分に取る必要がある．

```
> par(mar=c(5,4,5,4))
> plot(1:10)
> axis(side=4)
```

14.3　グラフィックスパラメータの一時的変更

この節で紹介するグラフィックスパラメータは，関数 par() で設定することが出来るし，関数 plot() や多くの高水準作図関数の引数としてグラフィックスパラメータを設定することも出来る．例えばプロット点の形を指定するパラメータ pch に関しては，以下の 2 通りの設定方法を用いることが出来る．ただし，関数 par() で設定した場合は，設定した後はもう一度パラメータ値を変更するまではそのままの設定値が使われるが，作図関数の引数としてグラフィックスパラメータを設定した場合は，そのときの作図の場合のみ設定値が使われる (それ以後は直前までの設定値が使われる)．

```
> plot(1:10, pch="+")      # 一時的にプロット点の形を "+" に変更する
> par(pch="+")             # これ以後，プロット点の形をずっと "+" に変更する
> plot(1:10)               # プロット点の形は "+" となる
> plot(1:20)               # これもプロット点の形は "+" となる
```

14.3.1　プロットに用いられる色

col と同様の命令で col.axis, col.lab, col.main, col.sub でそれぞれ軸，ラベル，タイトル，サブタイトルの色を指定することが出来る．

引数	機能
col = 1, col = blue	作図に用いる色 (color) を引数 col = 番号または色名で指定する (初期値は 1)．番号は 1 から順に「黒，赤，緑，青，水色，紫，黄，灰」となっている．
col = rgb(1, 0, 0), col = '#FFFFFF'	引数 col = rgb(赤，緑，青) で色を指定したり，16 進数で色を指定することも出来る．
col = rainbow(10)	以下の表の他に，rainbow(n) ，heat.colors(n) ，terrain.colors(n) ，topo.colors(n) ，cm.colors(n) を col に指定して色を決めることも出来る．
col = hsv(.5, .5, .5), col = gray(0.8)	それぞれ hsv 形式，グレースケールで色を決めることが出来る．
gamma = 1.0	ガンマ補正を行う．

14.3.2　プロットのマーカー

pch = 値 ，pch = "文字" で，点をプロットするときに用いるプロット文字を指定する．値は 0 から 25 の整数，文字はピリオドや "+" などで指定する．

pch	出力	pch	出力	pch	出力	pch	出力	pch	出力	pch	出力	pch	出力
0	□	1	○	2	△	3	+	4	×	5	◇	6	▽
7	⊠	8	✳	9	◇+	10	⊕	11	✡	12	⊞	13	⊗
14	⊠	15	■	16	●	17	▲	18	◆	19	●	20	•
21	○	22	□	23	◇	24	△						

14.3.3　テキスト・フォントに関するパラメータ

文字列に関するグラフィックスパラメータには以下のようなものがある．以下の引数によって文字の大きさや描画方向などを設定することが出来る．

引数	機能
adj = 0	テキスト文字列の調節 (0：左揃え，0.5：中心揃え，1：右揃え) を行う．また，adj = c(x,y) で x 軸および y 軸方向を別々に揃えることも出来る．
cex = 1	標準の大きさを 1 として，文字の拡大率を指定する．csi を変更すると，このグラフィックスパラメータもその値に応じて自動的に変化する．同様の命令で cex.axis, cex.lab, cex.main, cex.sub でそれぞれ軸，ラベル，タイトル，サブタイトルの拡大率を指定する．
ps = 20	テキストと記号の大きさをポイント単位で指定する．
text(x, y, labels="文字列", srt=90)	座標 (x, y) に labels で指定した文字列を表示する際に，文字の回転角を (単位は度で) 指定する (x 軸を基準とする) ．また，引数 crt では文字の回転角を (単位は度で) 指定する．
col = 1	色を指定する．同様の命令で col.axis, col.lab, col.main, col.sub でそれぞれ軸，ラベル，タイトル，サブタイトルの色を指定する．
font = 1	フォント番号を指定する (1：プレイン，2：ボールド，3：イタリック，4：ボールドイタリック, 5：シンボル文字 (Adobe symbol encoding)) ．同様の命令で font.axis, font.lab, font.main, font.sub でそれぞれ軸，ラベル，タイトル，サブタイトルのフォント番号を指定する．
family=""	フォント・ファミリーを指定する．デフォルトは "" で，デバイスのデフォルト値になるよう設定されている．他には "serif"，"sans"，"mono"，"symbol" が指定できる（デバイスの種類によっては無視される）．

14.3.4　線分の太さと形式

以下のパラメータが指定できる[*4]．

引数	機能
lwd = 1	線分の幅 (line width) を番号で指定する．値が大きいほど太い線になる．
lty=0，lty="blank"	線分の形式 (line type) を無し（透明の線）にする．
lty=1，lty="solid"	線分の形式 (line type) を実線にする．
lty=2，lty="dashed"	線分の形式 (line type) をダッシュにする．
lty=3，lty="dotted"	線分の形式 (line type) をドットにする．
lty=4，lty="dotdash"	線分の形式 (line type) をドットとダッシュにする．
lty=5，lty="longdash"	線分の形式 (line type) を長いダッシュにする．
lty=6，lty="twodash"	線分の形式 (line type) を二つのダッシュにする．
type = "p"	点プロット (デフォルト) を行う．
type = "l"	線プロット (折れ線グラフ) を行う．
type = "b"	点と線のプロットを行う．
type = "c"	"b" において点を描かないプロットを行う．
type = "o"	点プロットと線プロットの重ね書きを行う．
type = "h"	各点から x 軸までの垂線プロットを行う．
type = "s"	左側の値にもとづいて階段状に結ぶ．
type = "S"	右側の値にもとづいて階段状に結ぶ．
type = "n"	軸だけ描いてプロットしない（続けて低水準関数で作図する場合）．

[*4] 他にも 'lend'（線分の終端；0："round"，1："butt"，2："square"），'lheight'，'ljoin'（線分の結合形式；0："round"，1："mitre"，2："bevel"），'lmitre' が指定できる．

14.3.5　枠に関するパラメータ

引数	機能
bg = "blue"	背景の色を指定する．
fg = "blue"	前面の色を指定する．
bty = "o"	枠が箱型，四方が囲まれている形になる．
bty = "l"	枠が L 字型，左と下部だけに枠の線が引かれる．
bty = "7"	枠が 7 型，上部と右だけに枠の線が引かれる．
bty = "c"	枠が C 字型，右部を除き枠の線が引かれる．
bty = "u"	枠が u 字型，上部を除き枠の線が引かれる．
bty = "]"	枠が] 字型，左部を除き枠の線が引かれる．
bty = "n"	枠を描かない (箱を描く) ．

14.3.6　軸に関するパラメータ

引数	機能
ann = F	FALSE を指定すると，軸と全体のタイトルを描かなくなる．
las = 0	ラベルを各軸に並行して描く．
las = 1	ラベルをすべて水平に描く．
las = 2	ラベルを軸に対して垂直に描く．
las = 3	ラベルをすべて垂直に描く．
lab = c(5,5,7)	長さ 3 の数値ベクトルの最初の二つで x , y 軸につける目盛の数を指定する．最後の一つはラベルのサイズを指定するはずだが，今のところ (R 2.1.1) は移植されていない．
mgp = c(3,1,0)	長さ 3 の数値ベクトルでそれぞれ軸タイトル，軸ラベル，軸線が描かれる位置を枠から何行分 (mex 単位) 外側にするかを指定する．
tck = NA	軸の目盛線の長さを枠の大きさに対する割合で指定し，値が正ならば線が図の内側に，値が負ならば図の外側に目盛線が描かれる．ここで tck = 1 とすれば，図に格子を入れることが出来る．初期値は NA だが，これは tcl = −0.5 を用いるという意味である．
tcl = −0.5	軸の目盛線の長さをテキスト行の高さ (mex) の割合で指定する．正の値を指定すれば目盛を内側に向かって描き，負の値なら外側に描く．もし tcl = NA とすれば，tck に −0.01 がセットされる．
xaxt = "s" , yaxt = "s"	軸を描くか否かを指定する．この場合は通常の軸が使われていることを示す（"l" や "t" を指定しても同じ働き）．
xaxt = "n" , yaxt = "n"	軸を描くか否かを指定する．この場合は軸を描かない．xaxt = "n" と yaxt = "n" を同時に指定すると axes = F と同じになる．
xaxs = "r" , yaxs = "r"	x , y 軸のスタイルを指定する．まずデータ範囲を両側 4 % 広げ，次に範囲内でラベルがきれいに表示できる軸を見つける．
xaxs = "i" , yaxs = "i"	x , y 軸のスタイルを指定する．データ範囲内できれいに表示できる軸を見つける．
par(xaxp) , par(yaxp)	x，y 軸の目盛りの区切りを参照する．最初の二つが両端の目盛りの座標，最後の値がその内側の区切りの個数を表す．値自体を変更してもあまり意味が無い．
xlog = F , ylog = F	それぞれ対数 x 軸，対数 y 軸の使用を論理値で指定する．もし TRUE ならば対数軸が使われる．新しいデバイスに対しては既定値は FALSE に設定される．

第 15 章

対話的作図関数，画面分割，重ねた図

R では対話的作図関数を使って，マウスなどで図表の位置を示して座標を得たり，プロット点を追加したり除いたりすることが出来る．また，R では一つのデバイスに複数の図を描いたり，図を重ねて表示することが出来る．

15.1　座標値の取得：locator()

関数 locator() を使って，デバイス上の任意の位置の座標を知るすることが出来る．関数 locator() を実行すると，デバイス上に "＋" マーク（クロスヘア）が表示されるので，これをマウスで操作すればよい．マウスの左ボタンをクリックするとその点が入力される．Windows 版 R ならばマウスの中央のボタンをクリックすることによって入力を終了することが出来る．

```
> locator()              # 引数無しで使うと，入力終了操作が行なわれるまで座標値の入力をする
> locator(10)            # 10 点入力した時点で終了（既定値は 512 点まで）
```

locator() は入力した位置の座標 x と y を成分とするリストを返すので，マウスでプロットする際に，以下の様に命令すれば入力した位置の座標を保存することが出来る．

```
> z <- locator()         # z に入力した位置の座標 x と y を成分とするリストが付値
```

外れ値の近くに目印のテキストを置きたい場合は以下の様にすればよい．

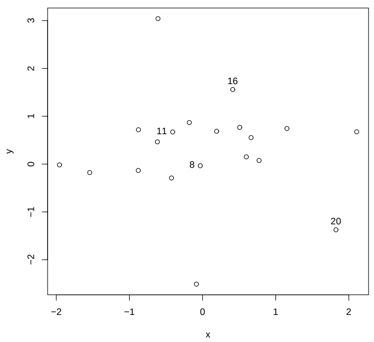
```
> text(locator(1), "Outlier", adj=0)
```

15.2　点の情報の取得：identify()

対話的に図表のある点の情報（座標，何番目にプロットされた点かを表す観測番号）を得るには関数 identify() を使用する．関数 locator() と同様，マウスで "＋" マークを操作し，情報を得たい点をクリックすればよい．

```
> x <- rnorm(20);  y <- rnorm(20)    # y に正規乱数を 20 個入れて
> plot(x, y)                         # (x, y) をプロットする
> identify(x, y, x)                  # マウスでクリックした観測点の x 座標が知りたいとき
> identify(x, y, y)                  # マウスでクリックした観測点の y 座標が知りたいとき
> identify(x, y)                     # マウスでクリックした観測点の観測番号が知りたいとき
[1]  8 11 16 20
```

identify() の第 3 引数には各点のラベルを指定することが出来，識別された点の近くに指定したラベルの座標が描かれる．この引数を省略すると座標ベクトル中の点の番号が描かれることになる．



identify() は，認識した点が x，y の何番目のデータであるかを表す観測番号が返される（上記の例では 8，11，16，20 番目の点が順に識別されている）．

```
> pt <- identify(x,y,y) # identify(x,y) でも identify(x,y,x) でも
> pt                    # pt には観測番号が格納される
[1]  8 11 16 20
```

(注意 1) plot=F と指定すればラベルを描き込まずに点の認識だけを行なう (認識された点の番号のベクトルは返される)
.

(注意 2) 例えばクリックした位置の周囲 0.25 インチ以内に点が無い場合や近くに既に指定済みの点がある場合は，入力された座標値に対して認識し得る点がなかったということで，エラー音と共に以下の様なメッセージが表示されてしまい，ラベルも表示されない．

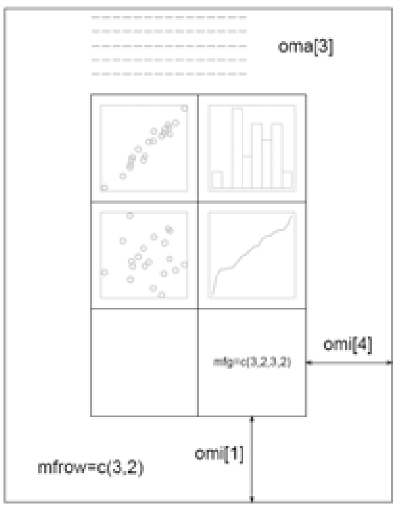
```
warning: no point with 0.25 inches
warning: nearest point already identified
```

15.3　画面の分割方法，重ねた図を描く方法

以下に出てくるパラメータの説明はグラフィックスパラメータと割り付けパラメータの節で扱っている．

15.3.1　画面（デバイス）を分割して図を描く

まず，画面（デバイス）の外周の余白を指定するパラメータがどこにあるかを紹介する．



タイトルや注釈テキストを入れる場合で，余白を指定する場合は par() の引数に以下のパラメータを入れて指定する．タイトルや注釈テキストを入れない場合は指定する必要は無い．ここで紹介する例では，全体のタイトルを描くため，まず oma で上部に 4 mex 分（mex の初期値は 1 なので 4 文字分）の余白を取っている．

```
> par(oma = c(0, 0, 4, 0))   # 下・左・上・右の順で余白を設定
```

画面（デバイス）を分割するパラメータは以下の様なものがある．

引数	機能
mfcol = c(m,n), mfrow = c(m,n)	画面を m 行 n 列に分割する．mfcol で指定した場合は列順に，mfrow で指定した場合は行順にグラフが描かれる．1 画面に戻す場合は mfrow=c(1,1) または mfcol=c(1,1) とすればよい．
mfg = c(i,j,m,n)	画面を m 行 n 列に分割している場合で，左上から順番に図を描きたくない場合に使う．次のグラフを i 行 j 列にを描く．
fig = c(4,9,1,4)/10	現在の画面における現在の図表の位置を指定するパラメータで，ページ内の任意の位置に図表をおくためのものである．値は百分率を指定し，「左側・右側・底辺・上側」の順に指定する．例の値はページの右下に図を置くように指定している．

ここでは画面を 2×2 に分割する．

```
> par(mfrow=c(2,2))
```

画面が分割できたら，グラフを 1 つずつ描けば，複数のグラフが（行順に）一度に表示される出力が得られる．

```
> plot(sin)
> plot(cos)
> plot(asin)
> plot(acos)
```

最後に outer = T と指定してから mtext を使うことで全体のタイトルを外周に書き込む．

```
> mtext(side = 3, line=1, outer=T, text = "Title", cex=2)
```

すると以下の左図のようになる．ここで最初の余白 (oma) を設定していなければ右図のようにタイトルがはみ出すことになる．

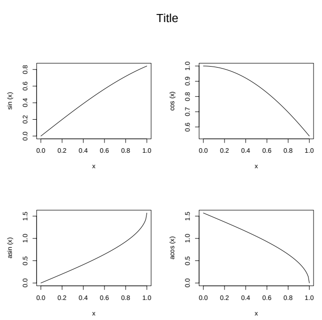


図 15.1　成功例

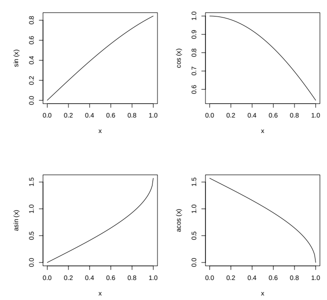


図 15.2　失敗例

(注意 1) 関数 frame() で図を消去すると図が消去されるが，このとき，図表番号の順番も 1 つ後に進んでいることに注意したい．

(注意 2) stars や pairs など，内部で複数図表を用いている高水準作図関数は，複数図表と同時に使うことは出来ない．

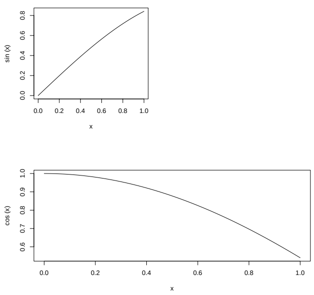
15.3.2　少し高度な画面（デバイス）分割

上で紹介したグラフィックスパラメータ mfrow や mfcol を用いることで画面を複数に分割することが出来るのだが，以下で紹介する関数を用いることで規則的な画面分割に限らず，自由に画面を分割することが出来るようになる．

layout() を用いた画面分割

関数 layout() を用いると，行列 mat で行数と列数を指定して画面を分割することが出来る．このとき画面は『行列の行数 × 行列の列数』に分割され，行列の成分が作図の順番となる．

```
> mat <- matrix(c(1,0,2,2), 2, 2, byrow = TRUE)
> mat
     [,1] [,2]             # このとき画面（デバイス）は
[1,]    1    0             # （ 1 番目の図）　（ここは空白のまま）
[2,]    2    2             # ( --------- 2 番目の図 --------- )
> layout(mat)              # と分割される
> plot(sin)
> plot(cos)
```



layout.show(n) でデバイス番号を確認することが出来，また lcm(x) で長さを指定することも出来る．

```
> layout(matrix(c(1,3,2,2), 2, 2, byrow = TRUE), respect=T, widths=lcm(5), heights=lcm(5))
> plot(sin)
> plot(cos)
> plot(acos)
```

split.screen() を用いた画面分割

関数 split.screen() に長さ 2 のベクトル c(m, n) を引数に与えることで画面を分割することも出来る[*1]．分割された画面には番号が振られ，その番号（画面）を指定して図を描くことができる．例えば，画面を上下 2 つに分割する場合は以下のようにする．

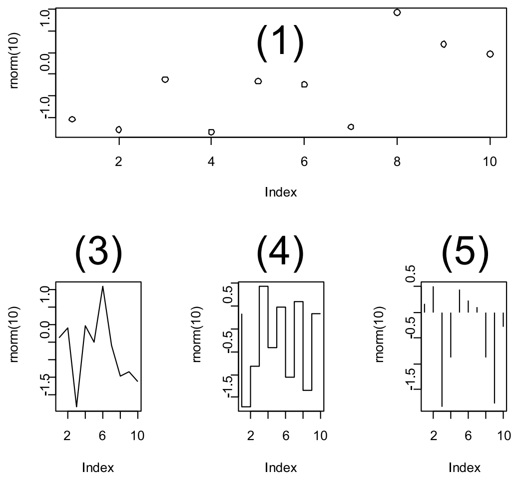
```
> split.screen(c(2,1))
[1] 1 2
```

この場合は上側が画面番号 1 ，下側が画面番号 2 になる．ここで引数 screen に分割する画面番号を指定することによって，その画面をさらに分割することも出来る．例えば上の例で出来た画面 2 をさらに 3 つに分割してグラフを描く．

[*1] 行列を引数にしても画面分割を行うことも出来る

```
> split.screen(c(1,3), screen = 2)
[1] 3 4 5
> screen(1); plot(rnorm(10))                # 画面 1 にプロット
> screen(3); plot(rnorm(10), type="l")      # 画面 3 にプロット
> screen(4); plot(rnorm(10), type="S")      # 画面 4 にプロット
> screen(5); plot(rnorm(10), type="h")      # 画面 5 にプロット
```

以下の出力結果で括弧つきの番号は画面番号を表す (実際の出力には表示されない)．



画面消去をする場合は関数 frame() で消去することが出来る．関数 erase.screen() は図を消去するのではなく，背景色で塗りつぶすだけなのでメモリを食うので使用は控えた方が良い．また，通常の 1 画面形式に戻る場合は，関数 close.screen(all=T) を実行すればよい．

15.3.3　重ねた図を描く

単純な作図ならば，引数に add=T を入れることで重ねた図を描くことができる．2 つのグラフの x 軸と y 軸の座標は自動的に合わせられる．

```
> plot(cos, -pi, pi, lty=2)
> curve(sin, add=T)
```

しかし，高水準作図関数の中には add=T を引数にもってくることが出来ないものがある．そこで，パラメータ new を指定することで，既存のグラフに新たなグラフを上書きするように指定することが出来る．例えば，以下のようにすることで 2 つのプロットを重ね合わせることが出来る．

(注意 1) par(new=T) で重ね描きする場合，そのままでは軸と軸のラベルが重ね描きされる．2 回目の plot() によって軸と軸のラベルが重ね書きされるのを避けるため，1 回目のプロット時に axes=F , xlab=””，ylab=””（もしくは ann=T ）を指定するのが得策である．

(注意 2) par(new=T) で重ね描きする場合，普通はそれぞれのグラフの座標範囲が異なるため，仕上がりがおかしくなる．重ね描きする全てのグラフが同じ座標範囲となるよう，全ての作図関数で xlim = c(x0, x1)，ylim = c(y0, y1) を指定すること．

```
x <- rnorm(100)                                        # 正規乱数を 100 個生成して
hist(x, xlim=c(-4,4), ylim=c(0,0.5), prob=T, ann=F)    # ヒストグラムを描く
par(new=T)                                             # 上書き可にしてから
plot(density(x), xlim=c(-4,4), ylim=c(0,0.5),
     xlab="", ylab="", main="", col="red")             # 密度推定の曲線を上書きする
```

この後，axis() で 2 回目のグラフ（plot(density(x)...)）が描くはずだった y 軸を図の右側の余白に描くことが出来る．

```
> axis(side=4)
```

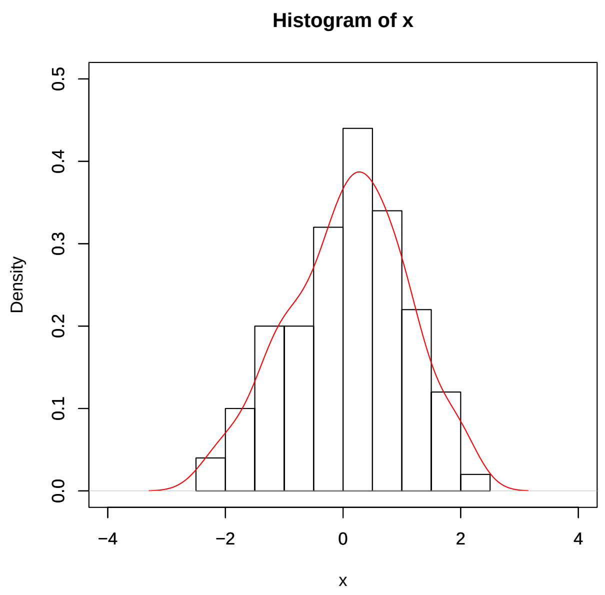


図 15.3　axis() 指定前

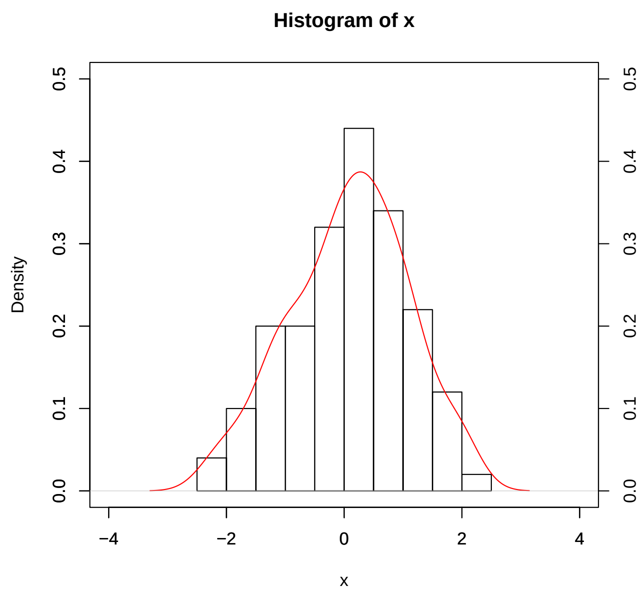


図 15.4　axis() 指定後

(注意) axis() で右側に y 軸を追記する場合，パラメータ mar で右側の余白を空けておいた方が良い．余白を空けないと不具合が生じることがある．

第 VI 部

統計解析篇

第 16 章

統計的処理の基本事項

16.1　基本統計量

四則演算や初等数学関数は既に扱ったので，ここでは基本的な統計量を求める関数を紹介する．まず，10 人 の患者から成る 2 グループ x，y にそれぞれ異なる睡眠薬を飲ませ，睡眠時間の増加を示すデータを準備する[*1]．

```
> x <- c(0.7,-1.6,-0.2,-1.2,-0.1,3.4,3.7,0.8,0.0,2.0) # グループ 1 の睡眠時間の増加を示すデータ
> y <- c(1.9, 0.8, 1.1, 0.1,-0.1,4.4,5.5,1.6,4.6,3.4) # グループ 2 の睡眠時間の増加を示すデータ
```

データを読み込ませた後は，次に紹介する関数で基本統計量を求めることになる[*2]．

関数	ave(x)	fivenum(x)	IQR(x)	max(x)	mean(x)	median(x)	min(x)
意味	平均 (因子)	5 数要約	4 分位偏差	最大値	平均	中央値	最小値
関数	quantile(x)	range(x)	sd(x)	sum(x)	var(x,y)	weighted.mean(x)	
意味	クォンタイル点	範囲	不偏標準偏差	総和	不偏分散	重み付け平均	

以下に例を示す．ちなみに，関数 summary() で要約統計量が得られる．

```
> quantile(x)                                       # x のクォンタイル点
    0%     25%     50%     75%    100%
-1.600 -0.175  0.350  1.700  3.700
> summary(y)                                        # y の要約統計量
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -0.100   0.875   1.750   2.330   4.150   5.500
```

16.1.1　度数分布表

度数分布表を描いた後，pie(result) や barplot(result) で円グラフや棒グラフを描くことが出来る．

```
> breaks <- seq(-2, 4, 2)                 # 度数分布表の区間を指定する
> ( result <- table(cut(x, breaks)) )    # 結果を result に代入
(-2,0]  (0,2]  (2,4]
     5      3      2
```

[*1] このデータについては 170 頁以降で詳しく扱う．

[*2] 5 数要約 fivenum() は「最小値・下側ヒンジ・中央値・上側ヒンジ・最大値」を，4 分位偏差 IQR() は「第 3 四分位から第 1 四分位を引いた値」を，クォンタイル点 quantile() は「最小値・第 1 四分位・中央値・第 3 四分位・最大値」を，範囲 range() は「最小値と最大値」を出力する．また，平方和，歪度及び尖度はそれぞれ sum((x-mean(x))^2)，mean((x-mean(x))^3)/(sd(x)^3)，mean((x-mean(x))^4)/(sd(x)^4) で求めることが出来，データが正規分布に近い場合は「歪度が 0 付近・尖度が 3 付近」になっている．パッケージ e1071 の中には，歪度と尖度を求める関数 skewness()，kurtosis() が用意されている．

16.1.2　標本分散と不偏分散，標本標準偏差と不偏標準偏差

データの不偏分散を求める関数 var() は不偏分散を求める関数であって，標本分散を求める関数ではないことに注意．すなわち，データ x のデータ数を n，平均を求める関数を E() とすると，var(x) は以下を求めている．

$$var(\mathrm{x}) = \frac{n}{n-1}\left[E(x^2) - \{E(x)\}^2\right]$$

よって，標本分散を求める場合は var() の結果を $(n-1)/n$ 倍する必要がある．標本標準偏差を求める場合も同様である．

```
> var(x)                                                      # 不偏分散
[1] 3.200556
> variance <- function(x) var(x)*(length(x)-1)/length(x)    # 標本分散を求める関数を定義
> variance(x)
[1] 2.8805
> sd(x)                                                       # 不偏標準偏差
[1] 1.789010
> sqrt(variance(x))                                           # 標本標準偏差
[1] 1.697204
```

16.1.3　分散共分散行列・相関行列

2 次元以上のデータに対する不偏共分散（または不偏共分散行列）も関数 var() で求めることが出来る．引数は 1 次元データを 2 つ与えても，行列を与えても，どちらでも良い．この関数 var() は不偏共分散，不偏共分散行列を求める関数であって，普通の共分散，共分散行列を求める関数ではないことに注意．よって，標本共分散を求める場合は var() の結果を $(n-1)/n$ 倍する必要がある．

また，相関係数（または相関行列）は関数 cor() で求めることが出来る．引数は 1 次元データを 2 つ与えても，行列を与えても，どちらでも良い．

```
> var(x, y)
[1] 2.848333
> cor(x, y)
[1] 0.7951702
```

ところで，偏相関行列は以下の関数を定義することで求めることが出来る．

```
> my.cor <- function(x) {
+   tmpcor <- cor(x)
+   if (det(tmpcor) != 0)  sol <- solve(tmpcor)
+   else { library(MASS);  sol <- ginv(tmpcor) }
+   d <- diag(sol)
+   sol <- -sol/sqrt(outer(d,d))
+   rownames(sol) <- paste("Var", 1:ncol(x))
+   colnames(sol) <- paste("Var", 1:ncol(x))
+   return(sol)
+ }
> sampledata <- matrix(c(1:9), ncol=3, byrow=T)
> my.cor(sampledata)
```

```
      Var 1 Var 2 Var 3
Var 1    -1    -1    -1
Var 2    -1    -1    -1
Var 3    -1    -1    -1
```

16.1.4　データの規準化

各列が変量となっているデータ行列 x の各変量の単位が異なる場合に，各変量を平均が 0，分散が 1 になるように変換することがある．これを規準化または標準化という．関数 scale(x) を用いることで，データ行列 x を標準化することが出来る．標準化は平均を 0 にするためのセンタリング (それぞれの変量からその変量の平均を引く)，分散を 1 にするためのスケーリング (それぞれの変量をその変量の標準偏差で割る) の 2 つの操作によって行なわれる[*3]．なお，センタリングを行なった前後で標準偏差は変わらない．

```
> scale(x)
            [,1]
 [1,] -0.02794842
 [2,] -1.31357592
 ................
[10,]  0.69871059
attr(,"scaled:center")
[1] 0.75
attr(,"scaled:scale")
[1] 1.789010
```

16.2　確率分布と乱数

R ではさまざまな理論分布の確率密度 $f(x)$，累積分布 $P(X \leq x)$，確率点 $\min\{x : P(X \leq x) > q\}$，及びその分布に従う乱数を求めることが出来る．

16.2.1　関数の使い方

後に紹介する確率分布に関する関数を使用する場合は，以下のような対応表を用いればよい．この表は，確率分布名が xxx である分布の確率点は “ qxxx ” で求めることが出来ることを表す．例えば t 分布ならば，この累積分布は pt，確率点は qt で求めることが出来る．

用途	関数名 (確率分布名：xxx)	説明
確率密度 (pdf)	dxxx(q)	q は確率点を表す．例えば確率分布名が norm ならば dnorm(q) となる．
累積分布 (cdf)	pxxx(q)	q は確率点を表す．例えば確率分布名が norm ならば pnorm(q) となる．
確率点 (quantile)	qxxx(p)	p は確率を表す．例えば確率分布名が norm ならば qnorm(p) となる．
乱数	rxxx(n)	n は生成する乱数の個数を表す．例えば確率分布名が norm ならば rnorm(n) となる．

[*3] 関数 scale() に center=F と指定することによってセンタリングを抑制することが出来，scale=F と指定することによってスケーリングを抑制することが出来る．

16.3　R に用意されている確率分布

R では以下の理論分布が用意されている．ただし，スチューデント化された分布は qtukey (確率点) と ptukey (累積分布) を求める関数のみ，多項分布は dmultinom（確率密度）と rmultinom（乱数）のみ，誕生日問題の分布（近似解）は pbirthday（一致確率）と qbirthday（一致確率に必要な観測数）のみしか用意されていない[*4]．

分布名	確率分布名	パラメータ
ベータ分布	beta	shape1, shape2, ncp
二項分布	binom	classes, coincident, prob
誕生日問題の分布	birthday	size, prob
コーシー分布	cauchy	location, scale
カイ二乗分布	chisq	df, ncp
指数分布	exp	rate
F 分布	f	df1, df2, ncp
ガンマ分布	gamma	shape, scale
幾何分布	geom	prob
超幾何分布	hyper	n, m, k
対数正規分布	lnorm	meanlog, sdlog
ロジスティック分布	logis	location, scale
多項分布	multinom	n, size, prob
負の二項分布	nbinom	size, prob
正規分布	norm	mean, sd
ポアソン分布	pois	lambda
ウィルコクソンの符号付順位和統計量の分布	signrank	m, n
t 分布	t	df, ncp
一様分布	unif	min, max
スチューデント化された分布	tukey	nmeans, df
ワイブル分布	weibull	shape, scale
ウィルコクソン順位和統計量の分布	wilcox	m, n

(例 1) 確率分布 binom についての書式は以下の通りである．引数 lower.tail には論理値を指定し，TRUE ならば確率は P[X ≤ x] で，FALSE ならば P[X > x] が返される[*5]．

```
dbinom(x, size, prob)                          # x : ベクトル, prob : 成功の確率
pbinom(q, size, prob, lower.tail = TRUE)       # q : quantile のベクトル
qbinom(p, size, prob, lower.tail = TRUE)       # p : 確率ベクトル
rbinom(n, size, prob)                          # n : 観測数,  size : 試行数
```

(例 2) 自由度 4 の t 分布において有意水準 0.05 で両側検定を行なう場合は qt(0.025, 4)，qt(0.975, 4) として確率点を求めれば良いことが分かる．

[*4] MASS パッケージには多変量正規乱数生成関数 mvrnorm() が，mvtnorm パッケージには多変量正規乱数生成関数 rmvt() か用意されている．mvtnorm パッケージには他にも多変量正規分布に関する関数 dmvnorm()，pmvnorm()，rmvnorm() が用意されている．

[*5] 引数には log, log.p というものもあり，TRUE を指定すると結果は対数を取ったものが返される．

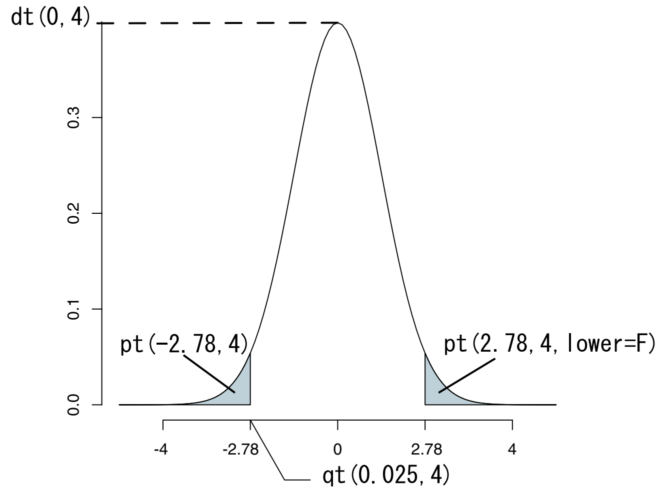


図 16.1　pt(), qt(), dt() の違い

(例 3) 正規分布 norm について確率密度 f(x)，その分布に従う乱数を求めるには以下の様にすればよい.

```
> dnorm(0)     # 確率密度（離散分布の場合は確率関数）
[1] 0.3989423
> rnorm(5)
[1] -2.0024918 -0.5996763 -0.3108348 1.2590405 0.3661534
```

(例 4) 確率密度のグラフを描くには以下のようにする.

```
> curve(dnorm, -4, 4, type="l")                          # 正規分布
> plot(0:10, dbinom(0:10, 10, 0.5), type="h", lwd=5)    # 二項分布
```

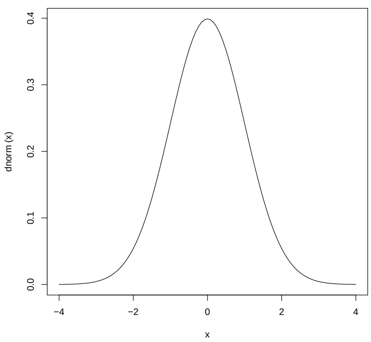


図 16.2　正規分布

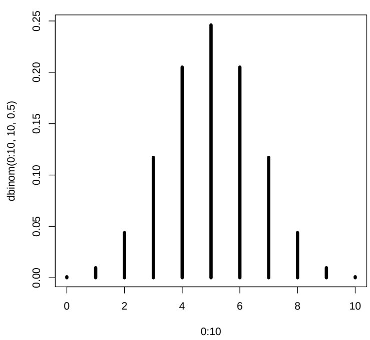


図 16.3　二項分布

(例 5) 自由度 (p, n - p - 1) の F 分布の上側 $100 \times \alpha\%$ 点は qf(1 - alpha, p, n - p - 1) で求められるので，例えば自由度 (2, 3) の F 分布の上側 5% 点は以下の様にして求めることが出来る.

```
> qf(0.95, 2, 3)           # qf(0.05, 2, 3, lower=F) でも可
[1] 9.552094
```

16.3.1　乱数の再現：set.seed()

疑似乱数は R 起動時に初期化され，毎回異なった乱数が発生される (ちなみに R は脅威のメルセンヌ・ツイスター法により一様乱数を生成している)．乱数を再現したい場合は，乱数の種を関数 set.seed() で指定すればよい．

```
> runif(5)                  # 一様乱数を 5 個生成すると
[1] 0.8039566 0.6593698 0.8120159 0.1279622 0.1115031
> runif(5)                  # 当然毎回違った乱数が得られる
[1] 0.5694816 0.4504077 0.1823374 0.4573758 0.9061499
> set.seed(101); runif(5)  # 乱数の種 (seed) を指定
[1] 0.37219838 0.04382482 0.70968402 0.65769040 0.24985572
> set.seed(101); runif(5)  # 乱数の種を同じにすれば乱数を再現することが出来る
[1] 0.37219838 0.04382482 0.70968402 0.65769040 0.24985572
```

16.3.2　2 次元正規乱数

相関が r だけある 2 次元正規乱数を生成する関数 r2norm() を定義する．

```
> r2norm <- function(n, mu, sigma, rho) {
+   tmp <- rnorm(n)
+   x   <- mu+sigma*tmp
+   y   <- rho*x + sqrt(1-rho^2)*rnorm(n)
+   return(data.frame(x=x,y=y))
+ }
> mydata <- r2norm(100, 0, 1, 0.5)
> cor(mydata$x,mydata$y)
[1] 0.5756098
```

16.3.3　3 変量分布の乱数

3 変量正規乱数

まず，3 変量正規分布に従う乱数を生成する関数 r3norm() を定義する．

```
> r3norm <- function(mu, A, n) {
+   U <- svd(A)$u;  V <- svd(A)$v;  D <- diag(sqrt(svd(A)$d))
+   B <- U %*% D %*% t(V)                                    # 行列 A の平方根
+   w <- c()
+   for (i in 1:n) w <- append(w, list(mu + B%*%cbind(rnorm(3))))
+   return(w)
+ }
> mu <- cbind(c(1,1,1))                                      # 平均ベクトル (縦ベクトル)
> A <- array(c(2,1,1,1,2,1,1,1,2), dim=c(3,3))
> w <- r3norm(mu, A, 2000)
```

3 変量 t 乱数

まず，以下を準備する．

R：　自由度 m のカイ二乗分布に従う確率変数
Z：　p 変量正規分布 N(0,I_p) に従う確率ベクトル（独立な標準正規乱数がズラッと縦に並んでいる）
V：　正定値対称行列（ちらばりを表す行列で分散共分散行列とは少し異なるもの）
V = C%*%t(C)：　コレスキー分解
X = sqrt(m/R)*Z：　p 変量楕円 t 分布 met(p,m,0,I_p) に従う
Y = μ+C%*%X：　p 変量楕円 t 分布 met(p,m,μ,V) に従う

次に，3 変量楕円 t 分布に従う乱数を生成する関数 met3() を定義する．

```
> met3 <- function(m, mu, V, n) {
+  # m  : 自由度，mu : 平均ベクトル，V : 散らばり行列，n  : 乱数の個数
+  U  <- svd(V)$u;  V1 <- svd(V)$v;  D  <- diag(sqrt(svd(V)$d))
+  B  <- U %*% D %% t(V1)
+  w <- c()
+  for (i in 1:n) {
+     R <- 0
+     for (j in 1:m) R <- R + rnorm(1)^2
+     w <- append(w, list(mu + B %*% (cbind(rnorm(3))*sqrt(m/R))))
+   }
+   return(w)
+ }
```

この分布の平均ベクトルは μ，分散共分散行列は $m/(m-2)*V$ 　($m>2$) なので，多変量楕円 t 乱数がうまく出来ているかどうかはこれで確かめることが出来る．

```
> mu <- cbind(c(1,1,1))
> V  <- array(c(2,1,1,1,2,1,1,1,2), dim=c(3,3))
> n  <- 1000
> w  <- met3(5, mu, V, n)
> sm <- 0
> for (i in 1:n) { sm <- sm + w[[i]] }
> sm <- sm/n
> sv <- 0
> for (i in 1:n) { sv <- sv + (w[[i]]-sm) %*% t(w[[i]]-sm) }
> sv <- sv/n
```

16.3.4　逆関数法による乱数の作り方

上に紹介している分布に無い分布に従う乱数を生成したくなる場合がある．まず，U を一様分布に従う確率変数とし，$X=F^{-1}(U)$ は

$$P(X\le x)=P(F^{-1}(U)\le x)=P(U\le F(x))=F(x)$$

となることより X は分布関数 F に従う．よって，分布関数の逆関数の引数に一様乱数を入れてやれば欲しい分布の乱数が得られる．

(例 1) 指数分布の分布関数 $F(x)=1-\exp(-x)$ の逆関数は $-\log(1-x)$ なので，$X=-\log(1-U)$ とすれば X は平均 1 の指数分布に従う．さらに，$1-U$ と U は同分布なので，$X=-\log(U)$ の U に一様乱数を入れてやることで平均 1 の指数乱数が得られる (ただし $x>0$)．

(例 2) ラプラス分布 (両側指数分布) の密度関数は $f(x)=1/2\exp(-|x|)$ だが，この場合は一様乱数 U_1, U_2 を生成し，U_1 が 1/2 以上ならば $-\log(U_2)$ を，U_1 が 1/2 未満ならば $\log(U_2)$ を採択すればラプラス分布に従う乱数が得られる．

例としてラプラス分布に従う乱数を生成する関数を定義する．

```
> rlaplace <- function(n) {
+   u <- log(runif(n))
+   v <- ifelse(runif(n)>1/2, 1, -1)
+   return(u*v)
+ }
> x <- rlaplace(1000)
```

16.3.5　棄却法による乱数の作り方

逆関数法は理論的には正しい方法なのだが，コンピュータは有限な値しか生成できないので，無限のサポートを持っている分布の裾の方の乱数の精度が悪くなる場合がある (例：逆関数法で生成した指数乱数)，そこで以下に棄却法による乱数生成の方法を紹介する．用いる関数を先に説明する．

$f(x)$ ： 生成したい乱数が従う分布の密度関数
$g(x)$ ： 乱数生成が容易な分布の密度関数
$c(\geq 1)$ ： $f(x) \leq c \cdot g(x)$ が成り立つ定数 (小さい方が良い)
$h(x)$ ： $= f(x)/\{c \cdot g(x)\}$

生成するアルゴリズムはこちら．

(1) u <– runif(1)
(2) v <– $g(x)$ に従う乱数
(3) u $\leq h$(v) ならば v を乱数として採択する

例として $f(x)=1/2\sin(x) \quad (0<x<\pi)$ という分布に従う乱数を生成することを考える．このとき

- $g(x)=1/\pi \quad (0<x<\pi)$
- $c=\pi/2$

とおくと $f(x) \leq c \cdot g(x)$ が成り立ち，この場合は $h(x)=\sin(x)$ となる．よって以下の関数 myrand (生成する乱数の数) で生成することが出来る．

```
> myrand <- function(x) {
+   y <- c();  i <- 1
+   while (i <= x) {
+     u <- runif(1);  v <- pi*runif(1);  w <- sin(v)
+     if (u < w){
+       y <- append(y, v);   i <- i+1
+     }
+   }
```

```
+   return(y)
+ }
> mydata <- myrand(1000)
```

16.3.6 周辺和を与えたランダムな 2 x 2 分割表

周辺和を与えて 2 x 2 分割表を関数 r2dtable() で作成することが出来る．引数は以下の通り．

n ： 生成する分割表の数
r ： 行和を与えるベクトル
c ： 列和を与えるベクトル (c、r の総和は一致する必要)

返り値は「行・列和がそれぞれ r,c であるランダムな 2×2 分割表 n 個のリスト」となる．

```
> x <- r2dtable(2, c(10,10), c(15,5))
> x
[[1]]
     [,1] [,2]
[1,]    8    2
[2,]    7    3
[[2]]
     [,1] [,2]
[1,]    6    4
[2,]    9    1
```

16.4 ランダム抽出・ブートストラップサンプリング

長さ n の与えられたベクトルの要素から長さ m の部分ベクトルをランダムに取り出す場合は，関数 sample() を用いればよい．デフォルトでは replace=FALSE となっている．

```
> n <- 20;  m <- 5
> x <- 1:n                           # 単に n としても良い
> sample(x, m, replace=TRUE)         # 同じ要素が選ばれても良い
[1]  5 18 20 18  5
> sample(x)                          # m = n でランダムな置換
[1]  6 11  9  2  1 14  4  7 15 20 16 12  3  5 18 17 13 10  8 19
> sample(c(0,1), 100, replace=TRUE)  # 100 個のベルヌーイ試行
 [1] 1 1 1 1 1 1 0 0 0 1 1 0 1 1 0 0 0 0 0 1 0 0 1 1 1 1 0 1 0 0 1 1 0 1 0 0 0
[38] 0 0 1 0 0 0 0 1 1 0 0 1 1 1 0 1 0 0 1 1 0 1 0 1 0 0 0 1 0 0 1 0 1 1 0 0 1
[75] 1 1 0 0 1 0 0 1 1 1 0 0 1 1 1 1 0 1 1 0 0 0 0 0 1 1
```

オプションで確率ベクトルを与えると，各要素が選ばれる確率を指定することが出来る (既定値は等確率 1/n)．

```
> p <- runif(n);  p <- p/sum(p)
> sample(x, m, replace=TRUE, prob=p)
[1] 17  4 16 12 20
```

第 17 章

データの分布とヒストグラム・密度推定

17.1　データの分布

まず，データの要素をグループ化するには，順序無し因子を生成する関数 factor() を使い，関数 table() を用いることで度数分布表を作ることが出来る[*1].

```
> data(warpbreaks)           # 1 台の織機当たりの反り破損の数
> attach(warpbreaks)         # 使うデータを固定
> fc <- factor(tension)      # 要素をグループ化
> levels(fc)                 # グループ化されているかを確認
[1] "L" "M" "H"
> table(fc)                  # グループ化したデータ fc に関する度数分布表
fc
 L  M  H
18 18 18
```

関数 cut() と組み合わせると，以下の様なことも出来る．

```
> tb <- factor(cut(breaks, breaks = 10+10*(0:6)))
> table(tb, fc)
         fc
tb        L M H
  (10,20] 3 6 8
  (20,30] 7 7 7
  .............
  (60,70] 2 0 0
```

また，データの平均や分散を計算する場合は関数 tapply() を用いる．tapply() の書式と例は以下の通り．

```
> # tapply(数値ベクトル，数値ベクトルと同じ長さの文字列ベクトル，適用したい関数)
> ( mymeans <- tapply(breaks, fc, mean) )     # breaks の平均をグループごとに計算
       L        M        H
36.38889 26.38889 21.66667
> ( mysd <- tapply(breaks, fc, sd) )          # グループごとに計算することも出来る
        L         M         H
16.446487  9.121009  8.352527
```

*1 他にも関数 xtabs() でクロス集計が出来る．

ところで，データの集合の要約は関数 summary() と fivenum() で，度数表示は stem() で知ることが出来る．stem() は，stem(データ, x) で幹の長さが既定値 (=1) の x 倍になる．

```
> stem(breaks)
  The decimal point is 1 digit(s) to the right of the |
  1 | 0234555667788899
  2 | 001111445666678889999
  3 | 00156699
  4 | 1234
  5 | 124
  6 | 7
  7 | 0
> summary(breaks)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  10.00   18.25   26.00   28.15   34.00   70.00
> fivenum(breaks)  # 最小値・下側ヒンジ・中央値・上側ヒンジ・最大値
[1] 10 18 26 35 70
```

しかし，1 次元データや 2 次元データならば視覚的に見た方がデータの特徴をつかみやすい場合が多い．そこでヒストグラムと密度推定の方法を以下で述べることにする．

17.2　ヒストグラムによる密度推定（準備）

ϕ を平均 -1，分散 1 の正規分布に従う確率変数，ψ を平均 2，分散 1 の正規分布に従う確率変数として

$$f(x) = 0.6\phi(x) + 0.4\psi(x)$$

となる密度から得られる乱数を生成する関数 generator() を定義する．

```
> truedensity <- function (x) {
+   0.6/sqrt(2*pi)*exp(-(x+1)^2/2) + 0.4/sqrt(2*pi)*exp(-(x-2)^2/2)
+ }
> curve(truedensity, xlim=c(-6,6), ylim=c(0,0.3), col=2)
```

次に，$f(x)$ に従う乱数を生成する関数を定義する．

```
> generator <- function(n) {
+   data1 <- rnorm(n)-1          # φ（x）に従う乱数
+   data2 <- rnorm(n)+2          # ψ（x）に従う乱数
+   data3 <- (runif(n) <= 0.6)  # 確率 0.6 で data1 を採択し，
+   data1*data3+data2*(1-data3) # 確率 0.4 で data2 を採択するための手続
+ }
```

関数 hist() でヒストグラムを表示することも出来る．区切り幅は R が適当に選択してくれる．probability=T と指定することによって相対度数で作図するように変更することが出来る．

```
> x <- generator(1000)
> hist(x)
```

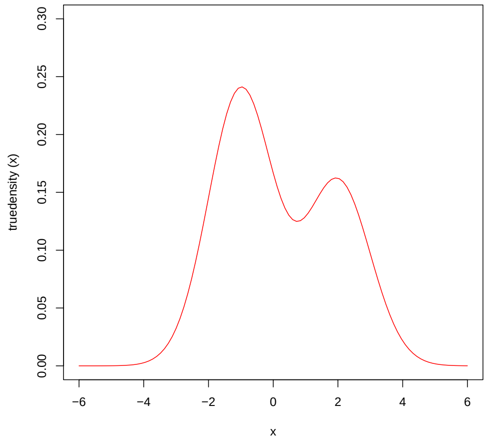


図 17.1　真の密度

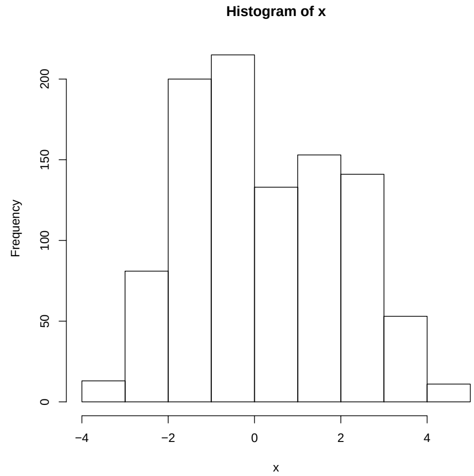


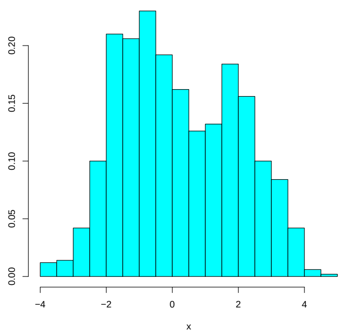
図 17.2　ヒストグラム

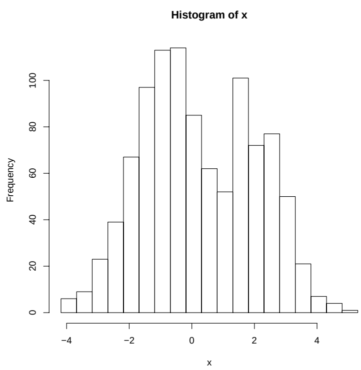
17.3　ヒストグラムによる密度推定

区切り幅は『適当に選択される』が『適切に選択される』わけではない．というのも，hist() のデフォルトは『データの範囲を $\log_2 n+1$ (n はデータの個数) 個の階級に分割して各階級に属するデータの数を棒グラフとして作図する』という Sturges (1926!!) の方法を用いているため，まず平滑化をし過ぎる嫌いがあり，さらにデータが正規分布 (正確には二項分布) から遠ざかれば遠ざかるほど当てはめが悪くなる．

そこでパッケージ MASS にある関数 truehist() (この関数では Scott (1992) が提唱した方法を用いている) や，パッケージ KernSmooth にある関数 dpih() (この関数では Wand (1995) が提唱した方法を用いている) を用いることで，より正確なヒストグラムを描くことが出来る．

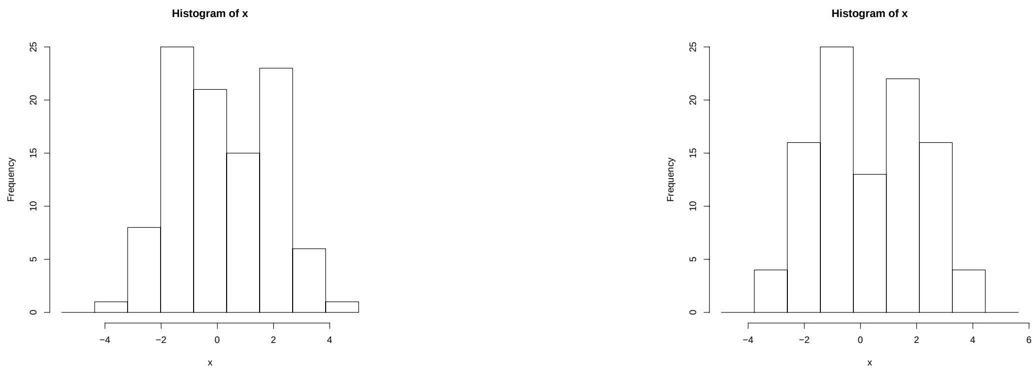
```
> library(MASS)
> x <- generator(1000)                      # hist(x, breaks="Scott") でも可
> truehist(x)                               # hist() の breaks には他に Sturges と FD を
                                            # 指定することが出来る
> library(KernSmooth)
KernSmooth 2.22 installed
Copyright M. P. Wand 1997
> x <- generator(1000)
> h <- dpih(x)                              # bin には区切り幅の点を表す x 座標を指定
> bins <- seq(min(x)-0.1, max(x)+0.1+h, by=h)  # bin が等差数列でなければ区切り幅の横幅
> hist(x, breaks=bins)                      # もバラバラな長さになる
```





ただし，ヒストグラムは左端の端点の位置によって形が変わってしまうという欠点がある (データ数が 50 未満の場合)
.

```
> library(KernSmooth)
> x <- generator(100)
> h <- dpih(x)
> bins <- seq(min(x)-2*h, max(x)+h, by=h)
> hist(x, breaks=bins)                              # 左下の図
> bins <- seq(min(x)-3*h/2, max(x)+3*h/2, by=h)
> hist(x, breaks=bins)                              # 右下の図
```



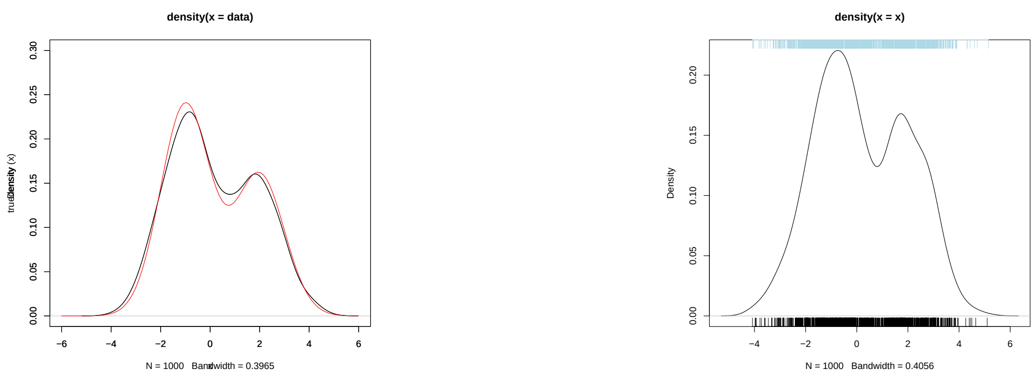
17.4 カーネルによる推定

前述のデータについて，関数 density() で密度推定することも出来る．この関数は与えられたカーネル関数とバンド幅を用いて密度関数推定を行う．ここでは，上のヒストグラムに上書きすることにする．

```
> data <- generator(1000)
> plot(density(data), xlim=c(-6,6), ylim=c(0,0.3))
> lines(density(data), xlim=c(-6,6), ylim=c(0,0.3))  # ここでは plot と同じ働き
> par(new=T)
> curve(truedensity, xlim=c(-6,6), ylim=c(0,0.3), col=2)
```

データに対応する x 軸上の点に縦線で印を付ける場合は以下のようにする．

```
> plot(density(x))
> rug(x)
> rug(jitter(x, amount = .1), side = 3, col = "light blue")
```



関数 density() の引数 bw には以下を指定することが出来る．

引数 bw	機能
nrd0	平滑化カーネル (ガウスカーネル) の標準偏差になるようにスケール化される．その既定値は標本数の 1/5 乗の 1.34 倍で割った標準偏差と四分偏差の小さい方を 0.9 倍したもの (= Silverman (1986) の『経験則』) でバンド幅を選択する．ただし四分偏差が 0 の場合は除かれ，このときは bw > 0 が保証される．(Silverman, B. W. (1986) Density Estimation. London: Chapman and Hall.)
nrd	nrd0 を Scott (1992) の方法により一般的したものを用いてバンド幅を選択．(Scott, D. W. (1992) Multivariate Density Estimation: Theory, Practice, and Visualization. Wiley.)
ucv	バイアス無しのクロスバリデーション規準によりバンド幅を選択.
bcv	バイアス付きのクロスバリデーション規準によりバンド幅を選択.
SJ-ste	パイロット評価を使用して (方程式を解くことにより) 帯域幅を選択する Sheather & Jones (1991) の方法でバンド幅を選択する．(Sheather, S. J. and Jones, M. C. (1991) A reliable data-based bandwidth selection method for kernel density estimation. Journal of the Royal Statistical Society series B, 53, 683-690.)
SJ-dpi	パイロット評価を使用して (プラグイン法により) 帯域幅を選択する Sheather & Jones (1991) の方法でバンド幅を選択する.

bw 以外に以下の引数が指定できる.

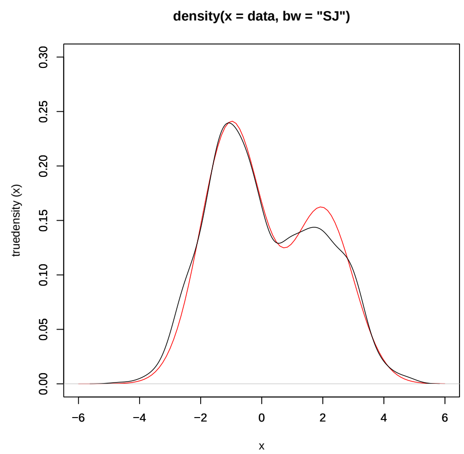
引数	機能
adjust	"bw" で指定されたバンド幅は，実際には adjust*bw となる．これは『デフォルトの半分』のようなバンド幅を指定する時に有用となる.
kernel , window	推定に用いるウインドウを指定する文字列で，"gaussian"（デフォルト），"rectangular" , "triangular" , "epanechnikov" , "biweight" , "cosine" , "optcosine" が指定できる．個人的には汎用性のある "epanechnikov" がお勧め．これらの文字列入力が面倒ならば window="g" などの様に先頭の一字に略しても良い．(注) "cosine"は S の命令であるが，これは "optcosine" よりも平滑化に優れる.
width	S 言語との互換性の為に存在する引数で，云うなれば "bw" である．width が与えられていて bw が与えられていなければ bw に width がセットされる.
give.Rkern	これを TRUE にすれば密度推定が行われず，代わりに選ばれたカーネルの canonical bandwidth が返される.
n	密度関数の値を求める等間隔点の数.
from , to	密度を from から to までの区間を n−1 等分した点で求める.
cut	推定された密度を，両端において 0 に下がらせるためのもの．左と右の値はデータの両極端を越えて「切られた」バンド幅となる.
na.rm	これを TRUE にすれば欠損値が x から取り除かれる．FALSE ならば，欠損値が見つかればエラーを起こす.

前で定義した generator() で $f(x)$ に従う乱数を 1000 個生成して密度推定した (バンド幅は "SJ" により選択)．真の密度を赤，density() で推定した密度を黒としてプロットしている[*2].

[*2] パッケージ KernSmooth にある関数 dpik() ならば，もっとデータに合わせた推定を行う．他にも関数 bkde() などが用意されている．

```
> library(KernSmooth)
> data <- generator(1000);  h <- dpik(data)
> est <- bkde(data, bandwidth=h)
> plot(est, type="l", xlim=c(-6,6), ylim=c(0,0.3))
```

```
> curve(truedensity, xlim=c(-6,6), ylim=c(0,0.3), col=2)
> par(new=T)
> plot(density(data,bw="SJ"), xlim=c(-6,6), ylim=c(0,0.3), xlab="", ylab="")
```

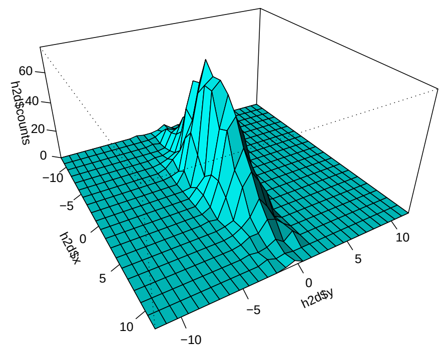


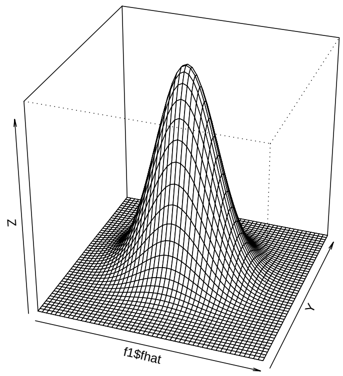
17.5　二次元データの密度推定

二次元データの当てはめを行う場合は，頻度ポリゴンを描く関数 hist2d() (パッケージ：gregmisc)，カーネル密度推定を行う関数 bkde2D() (パッケージ：KernSmooth)，関数 kde2d() (パッケージ：MASS) などが用意されている．

```
>  library(gregmisc)
Attaching package 'gregmisc':
        The following object(s) are masked from package:base :
        lowess
>  x <- rnorm(2000, sd=4)              # データ：無相関な 2 変量正規乱数
>  y <- rnorm(2000, sd=1)              # 遠近法プロット (persp) のためのデータを
>                                      # hist2d() を使用して作成
>  h2d <- hist2d(x, y, show=FALSE, same.scale=TRUE, nbins=c(20,30))
>  persp( h2d$x, h2d$y, h2d$counts,
+         ticktype="detailed", theta=60, phi=30,
+         expand=0.5, shade=0.5, col="cyan", ltheta=-30)

>  library(MASS)                       # 関数 width.SJ() を使うので MASS を呼び出し
>  library(KernSmooth)                 # 関数 bkde2D() を使うので KernSmooth を呼び出し
>  x <- rnorm(2000, sd=4)              # データの準備
>  y <- rnorm(2000, sd=1)              # データの準備
>  f1 <- bkde2D(cbind(x, y), bandwidth=c(width.SJ(x), width.SJ(y)))
>  persp(f1$fhat, phi = 30, theta = 20, d = 5)
```





17.6　lowess() による平滑化

関数 lowess() によってデータの平滑化を行うことも出来る．lowess() は平滑結果の座標を与える成分 x と y のリストを返す．平滑化は関数 lines() を用いて元の散布図に追加することが出来る．書式は以下の通り．

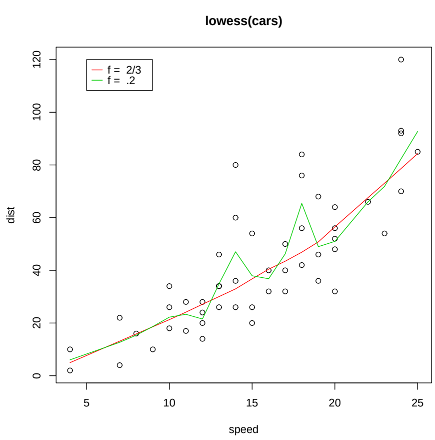
```
lowess(x, y, f=2/3, iter=3, delta=.01*diff(range(x)))
```

引数の説明は以下の通り[3]．

引数	機能
x , y	散布図中のプロットの座標を与えるベクトル．単一のプロット構造を指定しても良い．
f	平滑幅．これは各位置での平滑に影響を及ぼすプロットの点の割合を与える．値が大きい程より滑らかになる．
iter	実行されるべき頑健化繰り返しの数．小さな iter の値は lowess の実行を速くする．
delta	互いに距離 delta 以内に位置する x の値は lowess の出力で単一の値に置き換えられる．

以下に使用例を示す．

```
> data(cars)
> plot(cars, main = "lowess(cars)")
> lines(lowess(cars), col = 2)
> lines(lowess(cars, f = 0.2), col = 3)
> legend(5, 120, c(paste("f = ", c("2/3", ".2"))), lty = 1, col = 2:3)
```



[3] 他にも Nadaraya-Watson 推定による核関数を用いた回帰平滑化を行う関数 ksmooth() や，3 次の平滑化スプライン関数を当てはめる関数 smooth.spline()，Friedman の SuperSmoother を用いて平滑化を行う関数 supsmu() や，移動中央値平滑化を行う関数 runmed() , smooth() , smoothEnds() などがある．

第 18 章

検定手法のカタログ

この章では極々大雑把に検定関数の使用例を並べていく．

18.1　検定の準備

R には推定や検定を行う関数が多数用意されているが，そのほとんどが以下の様な使い方となっている．

```
> 関数名（データをベクトルや行列，データフレームなどで指定）
```

そして，データの指定方法は以下の通りである．

関数への指定方法	機能
関数名 (x)	1 つの標本について検定などを行う場合は 1 つのベクトル x を指定する．
関数名 (x, y)	2 つの標本について検定などを行う場合は 2 つのベクトル x, y を指定する．
関数名 (y ˜ x)	x にグループの種類，y にデータが入っている場合はこのように指定する．
関数名 (A)	2×2 分割表や分散分析表などについて検定を行う場合は，表の成分を行列 A で指定する．

例として，10 人の患者から成る 2 つのグループがあり，それぞれの患者に 2 つの睡眠薬を飲ませ，睡眠時間がどれだけ増加したかを示すデータ（データ名：sleep）を挙げる．

group1	0.7	−1.6	−0.2	−1.2	−0.1	3.4	3.7	0.8	0.0	2.0
group2	1.9	0.8	1.1	0.1	−0.1	4.4	5.5	1.6	4.6	3.4

```
> group1 <- c(0.7,-1.6,-0.2,-1.2,-0.1,3.4,3.7,0.8,0.0,2.0)      # グループ 1 の睡眠時間の増加
> group2 <- c(1.9, 0.8, 1.1, 0.1,-0.1,4.4,5.5,1.6,4.6,3.4)      # グループ 2 の睡眠時間の増加
```

ここでは，t 検定を行う関数 t.test() を用いてデータ sleep に関する検定を行う手順を紹介する．

18.1.1　検定例 (1)：一標本 t 検定

関数 t.test() で 1 標本 t 検定を行う場合の書式と引数を挙げる．一番目の引数にデータを与えればとりあえず検定が出来る．全ての引数を指定する必要は無い．を指定すれば検定が行われる．

```
> t.test(x, alternative=c("two.sided","less","greater"),
         mu=0, paired=FALSE, var.equal=FALSE, conf.level=0.95)
```

ここで，グループ 1 のデータについて一標本 t 検定を行う．この場合，母平均 mu が 0 であるかどうかを検定して，95% 信頼区間を求めている．

```
> t.test(group1, mu=0)

        One Sample t-test
data:  group1
t = 1.3257, df = 9, p-value = 0.2176
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 -0.5297804  2.0297804
sample estimates:
mean of x
     0.75
```

結果は有意ではない（母平均 mu が 0 でないとはいえない）．出力結果を解説する．

出力	機能
data	検定を行ったデータの名前．
t = 1.3257	t の値．これが棄却域に入っているかを見ている．
df = 9	t の自由度．
p-value = 0.2176	p 値．これが 0.05 より小さければ有意差がある（今は有意差が出ていない）．
alternative hypothesis	対立仮説（今は「母平均が 0 ではない」）．
95 percent confidence interval	95% 信頼区間．信頼区間は $[-0.5297804\ 2.0297804]$ となっている．
sample estimates	推定値．今は標本平均を推定していて 0.75 となっている．

今は両側検定を行ったが，片側検定を行うことも出来る．

```
> t.test(group1, mu=65, alternative="greater")

        One Sample t-test
data:  group1
t = -113.5692, df = 9, p-value = 1
alternative hypothesis: true mean is greater than 65
95 percent confidence interval:
 -0.2870553        Inf
sample estimates:
mean of x
     0.75
```

18.1.2　検定例 (2)：二標本 t 検定

「グループ 1 の睡眠増加時間の平均」と「グループ 2 の睡眠増加時間の平均」で差があるかどうかを検定する方法を紹介する．関数 t.test() で二標本 t 検定を行う場合の書式と引数を挙げる．

```
> t.test(x, y, alternative=c("two.sided","less","greater"),
         mu=0, paired=FALSE, var.equal=FALSE, conf.level=0.95)
```

関数 t.test() は以下の引数を指定することが出来る．

引数	機能
x	データ値の数値ベクトル.
y	追加のデータ値. 一標本について検定を行う場合は省略, 二標本検定を行う場合はデータを入れる.
alternative=”two.sided”	両側検定が行われる. alternative の指定を省略しても両側検定が行われる.
alternative=”greater”	より大きいかどうかの検定 (片側検定) .
alternative=”less”	より小さいかどうかの検定 (片側検定) .
mu	真の平均の値. 二標本検定の場合は平均の差を入力.
paired	論理値を入力. ここに TRUE を入れると対応のある t 検定をする.
var.equal	論理値を入力. ここに TRUE を入れると 2 つの標本の分散が等しいと仮定される.
conf.level	信頼度.

実際の検定はこちら. ここでは分散が等しいと仮定して, 関数 t.test() を用いて検定を行う[*1].

```
> t.test(group1, group2, var.equal=T)


        Two Sample t-test
data:  group1 and group2
t = -1.8608, df = 18, p-value = 0.07919
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.3638740  0.2038740
sample estimates:
mean of x mean of y
     0.75      2.33
```

p 値（p-value）は 0.07919 となっており 0.05 よりも大きい. よって, 結果は有意ではない（差があるとはいえない）. 出力結果を解説する.

引数	機能
statistic	t 統計量の値.
parameters	t 統計量の自由度.
p.value	検定の p 値. 特に重要. 有意水準 5% で検定している場合, この値が 0.05 よりも小さければ帰無仮説が棄却され有意とされる. 逆ならば帰無仮説が採択され, 有意ではないとされる.
95 percent confidence interval	平均の 95% 信頼区間.
estimate	一標本の場合は推定された平均の値. 二標本の場合は推定された平均の差.
null.value	一標本の場合は仮定された平均の値. 二標本の場合は仮定された平均の差.
alternative	対立仮説を表す文字列.
method	t 検定の種類を表す文字列.
data.name	データの名前を表す文字列.

[*1] 等分散を仮定しない場合はオプション var.equal=T を外せばよい. var.equal=F とした場合は「ウェルチの検定」を行う

18.2　一標本検定

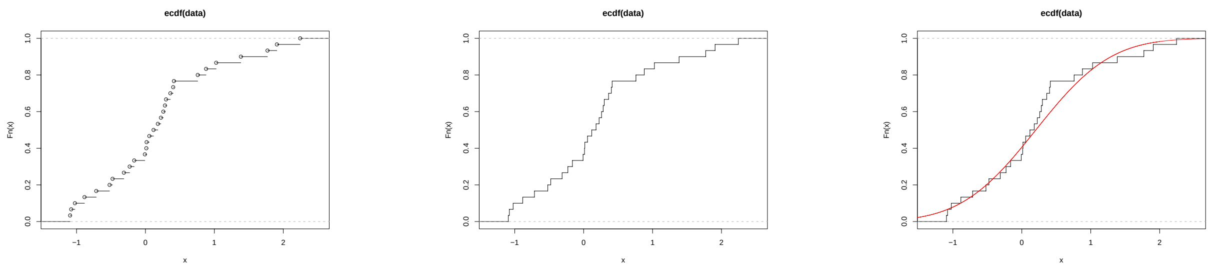
18.2.1　グラフによる正規分布との比較

パッケージ stepfun の中の関数 ecdf() で累積分布関数 (cdf) を描き，正規分布からのズレを見ることが出来る．stepfun は階段関数を扱うライブラリで，特に経験分布関数の作図ができる．

```
> data <- rnorm(30)                      # データを用意
> Fn <- ecdf(data)                       # ecdf() でデータフレームを作成
> Fn                                     # 中身を確認
> summary(Fn)                            # Fn の要約
> summary.stepfun(Fn)                    # Fn の詳しい要約
> plot(Fn)                               # 経験分布関数をプロット
> plot(Fn, do.point=F, verticals=T)      # 点を描かず，ジャンプ線（縦の線）を描く
```

この累積分布関数に正規分布の分布関数を上書きしてみる．

```
> x <- seq(-3,3,0.01)
> lines(x, pnorm(x, mean=mean(data), sd=sqrt(var(data))), col=2)
```



Q-Q プロットに関する関数も用意されている．

関数	機能
qqnorm(x)	x に対する期待正規ランクスコアをプロットする．データが正規分布に従っているかどうかを調べるために用いられ，qqnorm() によって描かれた散布図の点がほぼ直線上に並んでいればそのデータは正規分布に従っていると考えられる．
qqline(x)	上のプロットにデータの上四分位点と下四分位点を結ぶ直線を描く．
qqplot(x, y)	x の確率点に対する y の確率点 (Quantile-Quantile Plot：Q-Q プロット) を描く．

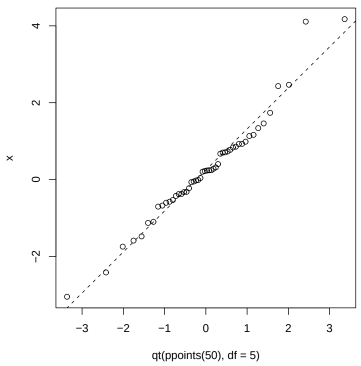
qqplot はプロットした点の座標（リスト）を返す．関数 lsfit() を使って回帰直線を求め，それを abline() を用いて重ね描きすることで点列についての当てはめ直線を描くことが出来る．

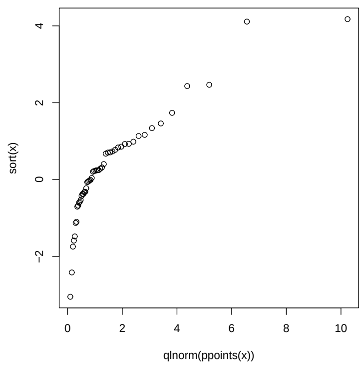
```
> x <- rt(50, df=5)
> xy <- qqplot(qt(ppoints(50), df=5), x)
> abline(lsfit(xy$x, xy$y), lty=2)
```

データが他の理論分布，例えば対数正規分布に従っているかどうかを調べるには以下の様にすればよい．x はデータ，ppoints() は QQ プロット用の確率ベクトルを作る関数，qlnorm() は対数正規分布の確率点を求める関数である[*2]．

```
> plot(qlnorm(ppoints(x)), sort(x))
```

[*2] qlnorm() の代わりに他の理論分布の確率点を求める関数を使えば，その分布に従っているかどうかを調べることも出来る．





18.2.2　正規性の検定

関数 shapiro.test() で Shapiro-Wilk 検定 を行うことが出来る．

```
> data(faithful)
> long <- faithful$eruptions[faithful$eruptions > 3]
> long           # データの確認
> shapiro.test(long)
        Shapiro-Wilk normality test
data:  long
W = 0.9793, p-value = 0.01052
```

関数 ks.test() で Kolmogorov-Smirnov 検定 を行うことが出来る．

```
> data(faithful)
> ( long <- faithful$eruptions[faithful$eruptions > 3] )       # データ出力は省略
> ks.test(long, "pnorm", mean=mean(long), sd=sqrt(var(long)))
        One-sample Kolmogorov-Smirnov test
data:  long
D = 0.0661, p-value = 0.4284
alternative hypothesis: two.sided
```

18.2.3　一標本検定・母平均の検定

詳しくは 170 頁を参照のこと．

18.2.4　一標本検定・対応のある二標本検定

躁鬱病の患者 9 人に対して精神安定剤による治療を行い，治療前 x と治療後 y に「憂鬱度」を測定した．以下はそのデータである．(数値が大きい方が症状が重い)

```
> x <- c(1.83,  0.50,  1.62,  2.48, 1.68, 1.88, 1.55, 3.06, 1.30) # first  visit
> y <- c(0.878, 0.647, 0.598, 2.05, 1.06, 1.29, 1.06, 3.14, 1.29) # second visit
```

検定の目的は「治療の前後で差があるかどうか」を検証することである．

対応のある二標本 t 検定

対応のある二標本 t 検定を行う場合は以下のようにする．

```
> t.test(x, y, paired = TRUE, alternative = "greater")
> t.test(y - x, alternative = "less")                # 上と同じ

        Paired t-test
data:  x and y
t = 3.0354, df = 8, p-value = 0.008088
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 0.1673028       Inf
sample estimates:
mean of the differences
              0.4318889
```

ウィルコクソンの符号付順位和検定

ウィルコクソンの符号付順位和検定を行う場合は以下のようにする．

```
> wilcox.test(x, y, paired = TRUE, alternative = "greater")
> wilcox.test(y - x, alternative = "less")                # 上と同じ

        Wilcoxon signed rank test
data:  x and y
V = 40, p-value = 0.01953
alternative hypothesis: true mu is greater than 0
```

18.3　二標本検定

170 頁のデータ sleep について，「『グループ 1 の睡眠増加時間』と『グループ 2 の睡眠増加時間』に差があるかどうか」を検証する．

二標本 t 検定（ウェルチの検定）

関数 t.test() でウェルチの検定（等分散を仮定しない場合）を行う．

```
> t.test(group1, group2)
        Welch Two Sample t-test
data:  group1 and group2
t = -1.8608, df = 17.776, p-value = 0.0794
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.3654832  0.2054832
sample estimates:
mean of x mean of y
     0.75      2.33
```

二標本 t 検定（等分散を仮定した場合）

「対応の無い二標本 t 検定」と呼ばれることもある*3．詳しくは 170 頁を参照のこと．ここでは，関数名 (y ~ x) という形式で検定を行う書式を示す．

```
> t.test(extra ~ group, data = sleep, var.equal=T) # extra：睡眠時間のデータ，group：群データ
```

ウィルコクソンの順位和検定・マン・ホイットニーの U 検定

関数 wilcox.test() でウィルコクソンの順位和検定を行う．マン・ホイットニーの U 検定とも呼ばれる．correct=F で連続性の修正を行わないよう指定することも出来る*4．

```
> wilcox.test(group1, group2)


        Wilcoxon rank sum test with continuity correction
data:  group1 and group2
W = 25.5, p-value = 0.06933
alternative hypothesis: true mu is not equal to 0
```

等分散性の検定：F 検定

170 頁のデータ sleep について，関数 var.test() で，group1 と group2 の母分散が等しいかどうかの検定を行う．

```
> var.test(group1, group2)


        F test to compare two variances
data:  group1 and group2
F = 0.7983, num df = 9, denom df = 9, p-value = 0.7427
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.198297 3.214123
sample estimates:
ratio of variances
         0.7983426
```

【参考】分散の均一性の検定：バートレットの方法

2 つのデータの分散の同等性は上で出てきたウェルチの方法で検定が行えた．データの組が 3 つ以上の場合はバートレットの方法で検定を行うことになる．

場所	葉の長さのデータ				
A	80	73	80	82	74
B	73	68	81	85	
C	85	93	88		

*4 t 検定が意味を持ち，最も威力を発揮する状況は，「2 群の分布が共に正規分布に近い」「分散がほぼ等しく，分布の違いが中心位置の違いに帰着できる」という状況である．
t 検定は外れ値があるようなデータには弱いため，そのような場合は外れ値に強い（ロバストな）検定であるウィルコクソンの順位和検定を用いるのが良い．

例えば，A，B，C の 3 つの場所のリンゴの葉の長さを調べてみたところ，次のようなデータが取られた．このとき，場所により葉の分散に差があると云えるかを有意水準 5% で検定する場合にバートレットの方法を用いる． 関数は bartlett.test() を用いる．

```
> length <- c(80, 73, 80, 82, 74, 73, 68, 81, 85, 85, 93, 88)   # データ
> group  <- c(rep(1, 5), rep(2, 4), rep(3, 3))                   # 群
> bartlett.test(length, group)

        Bartlett test of homogeneity of variances

data:  length and group
Bartlett's K-squared = 1.5451, df = 2, p-value = 0.4618
```

18.4 さまざまな検定

18.4.1 χ^2 (カイ二乗) 検定

適合度検定や独立性の検定は全て同じ関数 chisq.test() で行うことが出来る．この場合，引数のデータの形式と引数に確率を入れるかで検定内容が異なる．

検定したいデータの形式

以下の (1), (2) が適合度検定，(3) ～ (5) が独立性の検定である．いずれの場合も引数に correct=F を入れることで連続性の補正を抑制することが出来る．

(1) 1 **つのベクトルを引数にする場合** → c(8, 12, 10, 9, 5, 6) ： 例えばサイコロを振った結果を入れて，このサイコロが正しいサイコロであるかどうか (1~6 の生起確率が全て等しいのか) を検定する場合は 1 つのベクトルを引数にすればよい．

(2) 1 **つのベクトルと生起確率を引数にする場合** → c(20,8,5,2), p=c(4, 3, 2, 1)/10 ： 35 人の血液型を調べ，これより日本人の血液型は A，B，O，AB の順で 40%，30%，20%，10% の比率になっているかどうかを検定する場合は 1 つのベクトルと，その生起確率を引数にすればよい．

(3) 2 **列の行列を引数にする場合** → matrix(c(20, 15, 16, 4, 15, 7, 9, 4), ncol=2, byrow=T) ： 男女 90 人の血液型を調べて，男女の血液型の分布が同じかどうかを検定する場合は 2 列の行列を引数にすればよい．

(4) 2 **行** 2 **列の行列を引数にする場合** → matrix(c(20, 15, 16, 9), ncol=2, byrow=T) ： 男女 60 人に喫煙の有無を調べ，男女の喫煙率に差があるかどうかを検定する場合は 2 行 2 列の行列を引数にすればよい．

(5) m **列の行列を引数にする場合** → matrix(c(20, 15, ...), ncol=m, byrow=T) ： 例えば都道府県別に決まった人数を抽出して血液型を調べ，都道府県と血液型の間に関係があるかどうかを検定する場合は m 列 (m≥2) の行列を引数にすればよい．

実際の検定データ入力例

上のデータ形式それぞれの入力例である．

```
> chisq.test(c(8, 12, 10, 9, 5, 6))
> chisq.test(c(20,8,5,2), p=c(4, 3, 2, 1)/10)
> chisq.test(matrix(c(20, 15, 16, 4, 15, 7, 9, 4), ncol=4, byrow=T))
> chisq.test(matrix(c(20, 15, 16, 9), ncol=2, byrow=T))
```

```
> chisq.test(matrix(c(20, 15, 16, 4, 15, 7, 9, 4, 10), ncol=3, byrow=T))
```

18.4.2　カテゴリーデータの検定

χ^2 検定とフィッシャーの直接確率検定

例えば以下のようなデータが取られた場合に,『第一外国語の単位を取得する』ことと『第二外国語の単位を取得する』ことに関連があるかを検定することを考える.

		第一外国語		合計
		単位取得	単位取得不可	
第二外国語	単位取得	14	8	22
	単位取得不可	4	17	21
	合計	18	25	43

このような場合は，例えば χ^2 検定を用いればよい.

```
> x <- matrix(c(14, 8, 4, 17), ncol=2, byrow=T)
> chisq.test(x)

        Pearson's Chi-squared test with Yates' continuity correction
data:  x
X-squared = 7.0406, df = 1, p-value = 0.007968
```

χ^2 検定は近似的な方法である．正確な計算を行う場合は，フィッシャーの直接確率検定を用いる．関数は fisher.test() を使い，引数には 2×2 の行列を与えればよい.

```
> x <- matrix(c(14, 8, 4, 17), ncol=2, byrow=T)
> fisher.test(x)

        Fisher's Exact Test for Count Data
data:  x
p-value = 0.005089
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
  1.567890 39.549785
sample estimates:
odds ratio
  7.051895
```

フィッシャーの直接確率検定 (分割表が 2×2 よりも大きい場合)

上の場合は 2×2 の行列を引数に与えたが，分割表が 2×2 より大きい場合 (第二外国語を細分化する場合など) は，同じサイズの行列を引数に与えてフィッシャーの直接確率検定を用いればよい.

マクネマー検定

異なる状況下で同じ個体 (例えば学生) から 2 回データを取った場合など，関連のある (対応のある) 2 つの集団からデータが取られた場合に，半年の間に単位取得状況に変化があるかを検定する場合はマクネマー検定を用いる．関数は mcnemar.test() を使い，引数には 2×2 の行列を与えればよい．このとき，correct=F を引数に与えることで，連続

性の補正を抑制することが出来る．

		前期の英語		合計
		単位取得	単位取得不可	
後期の英語	単位取得	14	8	22
	単位取得不可	4	17	21
	合計	18	25	43

```
> x <- matrix(c(14, 8, 4, 17), ncol=2, byrow=T)
> mcnemar.test(x, correct=F)

        McNemar's Chi-squared test
data:  x
McNemar's chi-squared = 1.3333, df = 1, p-value = 0.2482
```

マクネマー検定 (分割表が 2×2 よりも大きい場合)

異なる状況下で同じ個体 (例えば学生) から 2 回データを取った場合など，関連のある (対応のある) 2 つの集団からデータが取られた場合に，調査項目が 3 つ以上上で，半年の間に単位取得状況に変化があるかを検定する場合もマクネマー検定を用いる．この時，引数には分割表と同じサイズの行列を与えればよい．

二群の比率の差の検定

「あなたは納豆が好きですか？」 という質問に対して，東京では 300 人中 245 人が，大阪では 250 人中 157 人が好きだと答えた場合に，好きな人の割合に差があるかどうか検定する場合は関数 prop.test() を使う．このとき，correct=F を引数に与えることで，連続性の補正を抑制することが出来る．[*5]．

```
> prop.test(c(245, 157), c(300,250), correct=F)

        2-sample test for equality of proportions without continuity  correction
data:  c(245, 157) out of c(300, 250)
X-squared = 24.6789, df = 1, p-value = 6.772e-07
alternative hypothesis: two.sided
95 percent confidence interval:
 0.1144583 0.2628750
sample estimates:
   prop 1    prop 2
0.8166667 0.6280000
```

符号検定と二項検定

ある納豆の製造会社が，大阪の人 10 人に「新製品『なっとうよん』を食べる前の納豆に対する評価」と「『なっとうよん』を 1 週間食べ続けてもらった後の納豆に対する評価」をアンケートで調査した．5 段階（5：良い ～ 1：悪い）で評価してもらった結果は以下の通りになった．

[*5] オッズ比とその信頼区間を求める場合はパッケージ vcd を呼び出した後で summary(oddsratio(table(A,B),log=F)) を実行すればよい．このとき，対数オッズ比を計算する場合は引数 log=F をつけないこと．さらに，いくつかの群の比率が独立変数と線形傾向があるかどうかを検定するための関数 prop.trend.test() がある．これはコクラン・アーミテージ検定といわれる検定方法である．指定は prop.trend.test (ベクトル 1, ベクトル 2) とすればよい．

(5：良い ～ 1：悪い)	A	B	C	D	E	F	G	H	I	J
食べる前	4	1	1	2	1	2	4	3	3	2
食べた後	2	1	3	1	5	1	5	3	3	1
前 − 後	−	0	+	−	+	−	+	0	0	−

1 週間で納豆の好き嫌いの割合に差があったかどうかを検定するときは符号検定を行う[*6].

```
> x <- c(4, 1, 1, 2, 1, 2, 4, 3, 3, 2)
> y <- c(2, 1, 3, 1, 5, 1, 5, 3, 3, 1)
> binom.test(c(length(x[x>y]), length(x[x < y])))

        Exact binomial test
data:  c(length(x[x > y]), length(x[x < y]))
number of successes = 4, number of trials = 7, p-value = 1
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.1840516 0.9010117
sample estimates:
probability of success
             0.5714286
```

カッパ係数

例えば，A さんと B さんが同じものを測定することを考える．A さんと B さんの評価の一致度を測る場合は κ (カッパ) 係数を用いる．

		A さん		
		良い	普通	悪い
B	良い	80	15	5
さ	普通	10	40	20
ん	悪い	5	10	25

κ (カッパ) 係数を計算する際は，パッケージ vcd を呼び出した後で関数 Kappa() を実行すればよい[*7].

```
> measure2 <- matrix(c(80, 15,  5,
+                       10, 40, 20,
+                        5, 10, 25), 3, byrow=T)
```

[*6] コインを多数回投げた結果について，表と裏の出る確率が等しくなっているかを検定する場合は，比率の同一性検定 prop.test(表が出た回数, 裏が出た回数, p = 0.5) を行う．これは近似による検定なので，正確な検定を行う場合は二項検定 binom.test() を行えばよい．

```
binom.test(c(257, 243), p=1/2)    # binom.test(表が出た回数，裏が出た回数，表が出る確率)
```

[*7] κ (カッパ) 係数に関連して，信頼性係数 ICC (Intraclass correlation coefficient) を求める場合は，パッケージ irr 中の関数 icc() を用いればよい．

```
> data(anxiety)
> icc(anxiety, model="twoway", type="agreement")
```

```
> Kappa(measure2)
               value        ASE       lwr       upr
Unweighted 0.4915254 0.05492153 0.3838812 0.5991696  # カッパ係数
Weighted   0.4736842 0.08145561 0.3140342 0.6333343  # 重みつきカッパ係数（不一致の程度も考慮）
```

18.4.3　相関係数と無相関検定

以下のような 2 組のデータ x，y に相関があるかどうかの検定を行う場合は関数 cor.test() を使う.

```
> x <- c(70, 72, 62, 64, 71, 76, 60, 65, 74, 72)
> y <- c(70, 74, 65, 68, 72, 74, 61, 66, 76, 75)
```

このとき，引数 method に次の文字列を入れることで検定方法を代えることが出来る.

"pearson"：　ピアソンの積率相関係数の無相関検定を行う.
"kendall"：　ケンドールの順位相関係数の無相関検定を行う.
"spearman"：　スピアマンの順位相関係数の無相関検定を行う.

上のデータ形式それぞれの入力例である.

```
> cor(x, y, method="spearman")           # 単なる相関係数
[1] 0.929878
> cor.test(x, y, method="pearson")       # 無相関かどうかの検定

        Pearson's product-moment correlation
data:  x and y
t = 8.5685, df = 8, p-value = 2.656e-05
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.7957471 0.9883181
sample estimates:
      cor   0.9496011
```

18.4.4　多群の検定

例えば，A，B，C の 3 つの場所のリンゴの葉の長さを調べてみたところ，次のようなデータが取られた.

場所	葉の長さのデータ				
A	80	73	80	82	74
B	73	68	81	85	
C	85	93	88		

```
> length <- c(80,73,80,82,74,73,68,81,85,85,93,88)    # データ
> group  <- c( 1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3)    # 群
```

平均値 ± 標準偏差のプロット

各群について，平均値 ± 標準偏差のプロットを行う場合は，パッケージ gplots 中の関数 plotmeans() を用いればよい[*8].

```
> library(gplots)
> plotmeans(length ~ group)
```

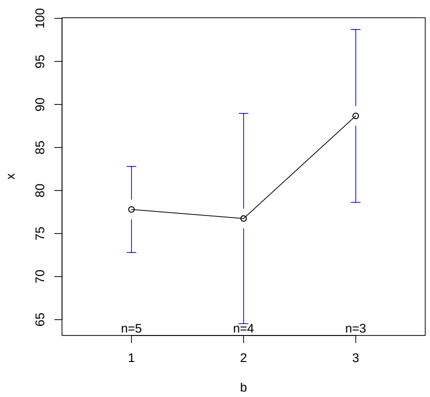


図 18.1　平均値 ± 標準偏差のプロット

各群の平均値と標準偏差が分かっている場合は，パッケージ gplots 中の関数 plotCI() を用いればよい.

```
> tmp   <- split(length, group)
> means <- sapply(tmp, mean)
> stdev <- sqrt(sapply(tmp, var))
> n     <- sapply(tmp, length)                # この length は関数の length() である
> ciw   <- qt(0.975, n) * stdev / sqrt(n)
> plotCI(x=means, uiw=ciw, col="black", barcol="blue", lwd=1)
```

一元配置分散分析

場所により葉の長さに差があるかどうかを検定する場合は関数 oneway.test() を用いる．このとき引数はベクトルで与える.

```
> oneway.test(length ~ group, var=T)                    # 一元配置分散分析


        One-way analysis of means


data:  length and group
F = 4.7828, num df = 2, denom df = 9, p-value = 0.03845
```

[*8] この場合はデータが 1 元配置となっているが，データが 2 元配置となっている場合は関数 interaction.plot() で同様のプロットが行える.

```
> attach(ToothGrowth)
> interaction.plot(dose, supp, len, fixed=TRUE)
```

クラスカル・ウォリス検定（対応の無い多群の差の検定）

場所により葉の長さに差があるかどうかを検定する場合は関数 kruskal.test() でも検定できる[*9].

```
> kruskal.test(length ~ group)          # クラスカル・ウォリス検定

        Kruskal-Wallis rank sum test

data:  length by group
Kruskal-Wallis chi-squared = 5.8408, df = 2, p-value = 0.05391
```

多重比較

全体で差があるかないかを見るのではなく，各群の平均値の大きさを多重比較することも出来る．例えば，pairwise.t.test() で，検定全体の有意水準が 5% を超えないよう調整した上で，多重比較を行うことが出来る．調整の方法は "holm" や "bonferroni" の方法が選択できる[*10].

```
> pairwise.t.test(length, group, p.adj = "bonf")

        Pairwise comparisons using t tests with pooled SD
data:  length and group
  1     2
2 1.000 -
3 0.074 0.059
P value adjustment method: bonferroni
```

フリードマン検定（対応がある多群の差の検定）

例えば，A，B，C の 3 つの場所について，肥料の量を変えてリンゴを育てていき，しばらく育てた後のリンゴの重さを調べてみたところ，次のようなデータが取られた.

場所＼肥料の量	10	20	30	40
A	180	193	200	201
B	150	189	186	190
C	144	178	166	173

肥料の効果があるかどうかを 5% の有意水準で検定する場合はフリードマン検定を行う.

```
> x <- matrix(c(180,193,200,201,
+               150,189,186,190,
+               144,178,166,173), ncol=4, byrow=T)
```

[*9] 以下のようにしても良い.

```
> x <- c(80, 73, 80, 82, 74)
> y <- c(73, 68, 81, 85)
> z <- c(85, 93, 88)
> kruskal.test(x, y, z)
```

[*10] 関数 pairwise.wilcox.test() を用いれば，ウイルコクソンの順位和検定で対比較を行うことが出来る．他にもテューキーの方法（関数 TukeyHSD()）やウイリアムズ検定（パッケージ multcomp 中の関数 simtest(x ~ b, type="Williams")）が用意されている.

```
> friedman.test(x)

        Friedman rank sum test
data:  x
Friedman chi-squared = 7, df = 3, p-value = 0.0719
```

最後に，ここまでで紹介できなかった関数を挙げる：
「ansari.test()：Ansari-Bradley 検定」「fligner.test()：Fligner-Killeen 検定」「mantelhaen.test()：Cochran-Mantel-Haenszel 検定」「mood.test()：Mood 検定」「quade.test()：Quade 検定」

18.5　検出力の計算と例数設計

R には以下の検出力計算用の関数が用意されている．

関数	機能
power.t.test()	t 検定の検出力を計算する
power.anova.test()	一元配置分散分析検定の検出力を計算する
power.prop.test()	比率の検定に対する検出力を計算する

t 検定の検出力（power）計算と例数設計を行う場合は，例数と Δ（想定した平均の差），標準偏差を指定すればその条件下での検出力が求まる．α（sig.level）や標本の種類（ type=c(”two.sample”, ”one.sample”, ”paired”)）, 両側か片側か（ alternative=c(”two.sided”, ”one.sided”)）を指定することも出来る．

```
> power.t.test(n=100, delta=1.0, sd=2.5)

     Two-sample t test power calculation
              n = 100
          delta = 1
             sd = 2.5
      sig.level = 0.05
          power = 0.8036466
    alternative = two.sided
 NOTE: n is number in *each* group
```

検出力（power）と Δ（想定した平均の差），標準偏差を指定すれば，その条件下で検出力を満たすための必要例数（1 群あたり）が算出される．

```
> power.t.test(power=0.9, delta=1.0, sd=2.5)

     Two-sample t test power calculation
              n = 132.3106
          delta = 1
             sd = 2.5
      sig.level = 0.05
          power = 0.9
    alternative = two.sided
 NOTE: n is number in *each* group
```

第 19 章

回帰分析

19.1　単回帰分析・直線回帰

関数 lm() と predict() を用いて単回帰分析を行うことができる．例えば以下のようなアメリカ桜のデータ trees があったとする．このデータに関して散布図を描いた後，単回帰分析を行う．

Girth：　木の直径（数値，単位はインチ）
Height：　木の高さ (数値，単位はフィート）
Volume：　長さ 1 フィートの立方体に切ったときの重さ

```
> plot(Volume ~ Girth, data = trees)                              # 散布図を描く
> result <- lm(Volume ~ Girth, data = trees)                      # 回帰分析を行う
> abline(result)                                                  # 推定回帰直線を描く
> summary(result)                                                 # 分析結果の要約
Call:
lm(formula = Volume ~ Girth, data = trees)
Residuals:
    Min      1Q  Median      3Q     Max
-8.0654 -3.1067  0.1520  3.4948  9.5868
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -36.9435     3.3651  -10.98 7.62e-12 ***
Girth         5.0659     0.2474   20.48  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 4.252 on 29 degrees of freedom
Multiple R-Squared: 0.9353,     Adjusted R-squared: 0.9331
F-statistic: 419.4 on 1 and 29 DF,  p-value: < 2.2e-16
```

結果は -36.9435 が切片項，5.0659 が傾きの推定値となっている．ここで，予測区間と信頼区間をプロットしてみる．

```
> new <- data.frame(Girth = seq(8, 24, 0.5))
> result.pre <- predict(result, new, interval="prediction")    # 予測区間
> result.con <- predict(result, new, interval="confidence")    # 信頼区間
> matplot(new$Girth, cbind(result.pre, result.con), lty=c(1,2,2,3,3), type="l")
```

プロット結果は以下のようになる．

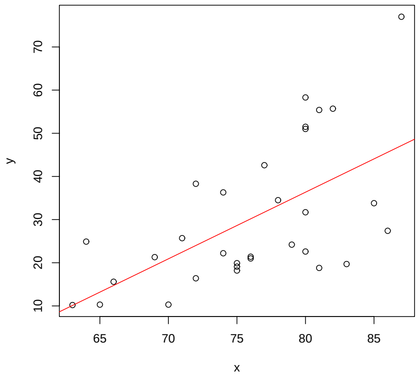


図 19.1　散布図と回帰直線

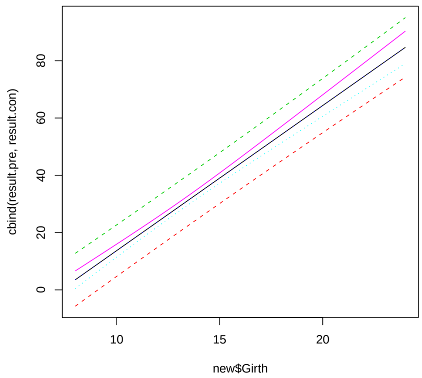


図 19.2　予測区間と信頼区間

19.2　関数 lsfit() による最小二乗法

関数 lsfit() により，y = X b + e の b を推定する．y には被説明変数のベクトルを与え，X には計画行列，すなわち説明変数 (それぞれがベクトル) を cbind したものを与える．つまり，この行列の各列が説明変量で各列が観測値となる．lsfit() の書式は以下の通り．

```
lsfit(x, y, wt=NULL, intercept=TRUE, tolerance=1e-07, yname=NULL)
```

引数の説明は以下の通り．

引数	機能
x	説明変数の行列.
y	被説明変数のベクトル.
wt	重みをつける場合はここに指定する.
intercept	デフォルトは TRUE で，この場合は自動的に計画行列（説明変数の行列）に定数項が加わる．定数項を加えたくない場合は FALSE を指定する.

例として，データ trees に関して最小二乗法を行う．

```
> x <- trees$Height;  y <- trees$Volume
> z <- lsfit(x, y)      # 推定結果を z に格納
> print(z)              # coefficients, residuals, intercept, qr を出力
```

出力の説明は以下の通り．

引数	機能
coef	回帰係数ベクトル．intercept=T (デフォルト) ならば第 1 要素が回帰式の定数項とる.
residuals	残差ベクトル.
intercept	引数 intercept と同じ値で，説明行列が定数項に対応する列を含んでいるか否かを示す.
qr	x に定数項に対応する列を追加した行列を QR 分解した結果．この要素の副要素 qt は t(Q)% *% y を表し (t(Q) は Q の転置行列) ，この要素の qt 以外の副要素は関数 qr が返す要素をそのまま保持している.

回帰による予測値を求める場合は，出力 z について以下の様に入力すればよい[*1]．

[*1] 関数 ls.diag(z) で残差の標準偏差やハット行列 H の対角成分などが得られる．

```
> y - z$residuals
 [1] 20.91087 13.19412 10.10742 23.99757 37.88772 40.97442 14.73747 28.62762
 [9] 36.34437 28.62762 34.80102 30.17097 30.17097 19.36752 28.62762 27.08427
[17] 44.06112 45.60447 22.45422 11.65077 33.25767 36.34437 27.08427 23.99757
[25] 31.71432 37.88772 39.43107 36.34437 36.34437 36.34437 47.14782
```

また，重相関係数 R = (予測値の分散) / (被説明変数の分散) を求める場合には以下の様に入力すればよい．

```
> sqrt(var(y - z$residuals)/var(y))
[1] 0.5982497
```

以下のように lsfit() の結果を abline() に渡すことによって，回帰直線を描くことも出来る．

```
> plot(x, y)
> abline(z, col=2)
```

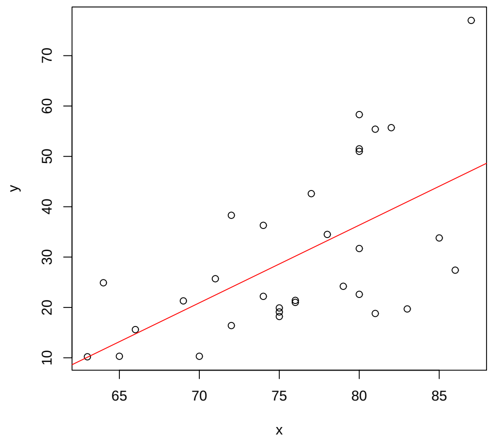


図 19.3　lsfit() による回帰直線

19.3　回帰分析と重回帰分析

関数 lm() により線形モデルの当てはめを行うことが出来る．この関数により，回帰分析や分散分析，そして共分散分析を行うことが出来る[*2]．

19.3.1　関数 lm() の書式と引数

書式は以下の通りである．重要なのはモデルを指定する引数 formula と，データ名を指定する引数 data である．

```
lm(formula, data, subset, weights, na.action,
   method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
   singular.ok = TRUE, contrasts = NULL, offset = NULL, ...)
```

[*2] 詳しい解説は『工学のためのデータサイエンス入門』(間瀬・神保・鎌倉・金藤 共著，数理工学社) を参照のこと．分散分析や非線形回帰についても詳しい解説が載っている．

引数 formula と回帰のモデル式の対応表を挙げる．

formula	回帰におけるモデル式
y ~ x	モデル式 $y = a + bx + \epsilon$（ ϵ は誤差項）について，目的変数 y と説明変数 x をベクトルで指定する．
y ~ x1 + x2	モデル式 $y = a + b_1x_1 + b_2x_2 + \epsilon$ （ ϵ は誤差項）について，目的変数 y と説明変数 x_1, x_2 をベクトルで指定する．
y ~ x1 * x2	交互作用項を含んだモデル式（ x1 : x2 でもよい）$y = a+b_1x_1+b_2x_2+b_3x_1x_2+\epsilon$（ ϵ は誤差項）について，目的変数 y と説明変数 x_1, x_2 をベクトルで指定する．
y ~ x1 + x2 + x1 * x2	上と同じ交互作用モデル．
y ~ (x1 + x2)^2	上と同じ交互作用モデル．
y ~ x − 1	切片 (定数) 項を除外する（ + 0 でも可）．
y ~ 1 + x + I(x^2)	多項式回帰：$y = b_0 + b_1x + b_2x^2 + \epsilon$．I(x^2) は poly(x,2) でも可．
y ~ x | z	z で条件付けしたときの y の x への単回帰．
y ~ . , data = データ名	あるデータに目的変数 y と説明変数 $x_1, \cdots$ が入っている場合で，モデル式が $y = a + b_1x_1 + \cdots + \epsilon$ （ ϵ は誤差項）である場合は，まず目的変数 y をベクトルで指定し，右辺は「y 以外」という意味で .（ピリオド）を指定することも出来る．

19.3.2　モデル情報を取り出す関数

lm() の返り値は当てはめられたモデルのオブジェクトとなっている．このオブジェクト obj を以下の関数に与えることで，新たな統計処理を行うことが出来る．この中からよく使うものを説明する．

関数	機能
AIC(obj)	AIC（赤池情報量規準）を求める．
anova(obj1 , obj2)	モデルを比較して分散分析表を生成する．
coefficients(obj)	回帰係数 (行列) を抽出．coef(obj) と省略できる．
deviance(obj)	重みつけられた残差平方和．
formula(obj)	モデル式を抽出．
logLik(obj)	対数尤度を求める．
plot(obj)	残差，当てはめ値などの 4 種類のプロットを生成．
predict(obj, newdata=data.frame)	提供されるデータフレームが元のものと同じラベルを持つことを強制する．値は data.frame 中の非ランダムな変量に対する予測値のベクトルまたは行列となる．
print(obj)	オブジェクトの簡略版を表示．
residuals(obj)	適当に重みつけられた残差 (の行列) を抽出する．resid(obj) と省略できる．df.residuals(obj) も参照されたい．
step(obj)	階層を保ちながら，項を加えたり減らしたりして適当なモデルを選ぶ．この探索で見つかった最大の AIC (赤池情報量規準) を持つモデルが返される．
summary(obj)	回帰分析の完全な要約が表示される．

19.3.3 回帰分析の例

(例 1) 以下は単回帰モデルの当てはめを行う[*3].

```
> result <- lm(Volume ~ Girth, data = trees)                  # 回帰分析を行う
> summary(result)                                             # 分析結果の要約
```

(例 2) 両対数を取ってから線形回帰モデルの当てはめを行うこともできる.

```
> result1 <- lm(log(Volume) ~ log(Girth), data = trees)
> summary(result1)
```

(例 3) 関数 anova() で分散分析表を書くことが出来る.

```
> anova(result1)
Analysis of Variance Table
Response: log(Volume)
           Df Sum Sq Mean Sq F value    Pr(>F)
log(Girth)  1 7.9254  7.9254  599.72 < 2.2e-16 ***
Residuals  29 0.3832  0.0132
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
> aov(result1)
Call:
   aov(formula = result1)
Terms:
                log(Girth) Residuals
Sum of Squares    7.925446  0.383244
Deg. of Freedom          1        29
Residual standard error: 0.1149578
Estimated effects may be unbalanced
```

(例 4) 以下はモデルの情報を取り出す一つの例である．以下は, result1 で使ったモデルに, さらに説明変数 log(Height) を取り入れて新たなモデルの当てはめを行っている．説明項に多少の変更を加えた場合でモデルの当てはめをやり直す場合には，この関数 update() を用いるのが便利である.

```
> result1 <- lm(log(Volume) ~ log(Girth), data = trees)
> result2 <- update(result1, ~ . + log(Height), data = trees)
> summary(result2)
```

[*3] 関数 lm() で回帰分析を行った要約結果を変数 result に代入した場合，result$fstatistic は実際の F 検定結果と自由度は返すが p 値は返さない．自力で F 統計量の p 値を計算する場合は以下のようにすればよい.

```
> result <- summary( result <- lm(Volume ~ Girth, data = trees) )
> f.stat <- result$fstatistic
> p.value <- 1-pf(f.stat["value"],f.stat["numdf"],f.stat["dendf"])
> p.value
value
    0
```

```
Call:
lm(formula = log(Volume) ~ log(Girth) + log(Height), data = trees)
Residuals:
      Min        1Q    Median        3Q       Max
-0.168561 -0.048488  0.002431  0.063637  0.129223
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -6.63162    0.79979  -8.292 5.06e-09 ***
log(Girth)   1.98265    0.07501  26.432  < 2e-16 ***
log(Height)  1.11712    0.20444   5.464 7.81e-06 ***
```

(例 5) 関数 step() を用いて，result2 のモデルについて，モデルから変数を 1 つ削った場合と変数を削らない場合の解析を全通り行い，各結果の AIC を比較している．

```
> result3 <- step(result2)
Start:  AIC= -152.69
 log(Volume) ~ log(Girth) + log(Height)

              Df Sum of Sq      RSS      AIC
<none>                        0.185 -152.685
- log(Height)  1     0.198    0.383 -132.185
- log(Girth)   1     4.628    4.813  -53.743
> summary(result3)
Call:
lm(formula = log(Volume) ~ log(Girth) + log(Height), data = trees)
Residuals:
      Min        1Q    Median        3Q       Max
-0.168561 -0.048488  0.002431  0.063637  0.129223



Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -6.63162    0.79979  -8.292 5.06e-09 ***
log(Girth)   1.98265    0.07501  26.432  < 2e-16 ***
log(Height)  1.11712    0.20444   5.464 7.81e-06 ***
```

(例 6) 説明変数に非説明変数以外の全てを指定する場合は .（ピリオド）を指定し，切片項を取り除く場合は −1 を指定する．

```
> result <- lm(log(Volume) ~ .-1, data = trees)
> summary(result)
Call:
lm(formula = log(Volume) ~ . - 1, data = trees)
Residuals:
      Min        1Q    Median        3Q       Max
-0.188133 -0.072707 -0.001769  0.062948  0.174312
```

```
Coefficients:
       Estimate Std. Error t value Pr(>|t|)
Girth  0.144695   0.006381   22.68  < 2e-16 ***
Height 0.017830   0.001138   15.66 1.09e-15 ***
```

(例 7) パッケージ wle を入れることで，関数 mle.cp()，mle.aic()，mle.cv() を用いることによりそれぞれ Cp 統計量，AIC，クロス・バリデーション規準により重回帰モデルの選択を行う．

```
> result <- mle.cp(log(Volume) ~ log(Girth) + log(Height), data = trees)  # cp 統計量
> result <- mle.cv(log(Volume) ~ log(Girth) + log(Height), data = trees)  # CV
> result <- mle.aic(log(Volume) ~ log(Girth) + log(Height), data = trees) # AIC
> summary(result)
Call:
mle.aic(formula = log(Volume) ~ log(Girth) + log(Height), data = trees)
Akaike Information Criterion (AIC):
     (Intercept) log(Girth) log(Height)      aic
[1,]           1          1           1  -64.560
[2,]           1          1           0  -36.700
[3,]           0          1           1    2.196
[4,]           0          1           0  169.000
[5,]           1          0           1  632.100
[6,]           0          0           1  976.300
[7,]           1          0           0 1158.000
Printed the first  7  best models
```

パッケージ wle を入れることで，関数 wle.cp()，wle.aic()，wle.cv() を用いることによりそれぞれ 重み付けされた Cp 統計量，AIC，クロス・バリデーション規準により重回帰モデルの選択を行うことも出来る．

19.4　一般化線形モデル

関数 glm() を用いることで一般化線形モデルを扱うことが出来る[*4]．一般化線形モデルのクラスは，応答変数の分布が正規分布（gaussian），二項分布（binomial），ポアソン分布（poisson），逆正規分布（inverse.gaussian），ガンマ分布（Gamma），そして応答分布がはっきりしないときのための擬似尤度モデル（quasi）を備えており，引数 family で指定することが出来る．ファミリー名とリンク関数の表を以下に挙げる．

引数 family	リンク関数
binomial	logit, probit, log, cloglog
gaussian	identity, log, inverse
Gamma	identity, inverse, log
inverse.gaussian	1/mu^2, identity, inverse, log
poisson	identity, log, sqrt
quasi	logit, probit, cloglog, identity, inverse, log, 1/mu^2, sqrt

ロジスティックモデル $p(x) = \dfrac{1}{1+\exp\{-(\beta_0+\beta_1 x)\}}$ の例を挙げる．

[*4] 関数 termplot() も参照されたい．ところで，線形混合モデルはパッケージ nlme 中の関数 lme() で実行できる．

```
> b0 <- -3;    b1 <- 0.5;    x <- seq(0, 10, 0.1)
> p  <- 1/(1+exp(-b0+b1*x))
> mydata <- rbinom(length(x), 1, prob=p)
> result <- glm(mydata ~ 1+x, binomial(logit))
> summary(result)
Call:
glm(formula = mydata ~ 1 + x, family = binomial(logit))
Deviance Residuals:
     Min        1Q    Median        3Q       Max
-0.28951  -0.16852  -0.09972  -0.05892   2.86202
Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -3.1513     1.4901  -2.115   0.0344 *
x            -0.4216     0.4925  -0.856   0.3919
```

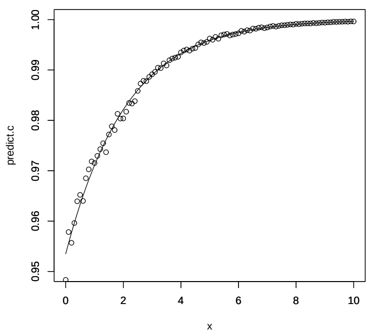
19.5　非線形回帰分析

関数 nls() を用いることによって非線形回帰分析を行うことが出来る[*5]..

```
> f <- function(x) 1/(1+exp(-3-0.5*x))
> x <- seq(0,10,0.1);   y <- f(jitter(x, amount=0.2))
> ( result <- nls(y ~ a/(1+exp(-b-c*x)), start=c(a=2,b=1,c=0.1)) )
Nonlinear regression model
  model:  y ~ a/(1 + exp(-b - c * x))
   data:  parent.frame()
        a         b         c
1.0000121 3.0203203 0.4931959
```

データに対して当てはめ曲線を描くことも出来る.

```
> predict.c <- predict(result)
> plot(x, y, ann=F, xlim=c(0,10), ylim=c(0.95,1));   par(new=T)
> plot(x, predict.c, type="l", xlim=c(0,10), ylim=c(0.95,1))
```



[*5] 非線形混合モデルはパッケージ nlme 中の関数 nlme() で実行できる.

第 20 章

その他の分析方法の紹介

詳しくは (その関数のパッケージを library() で呼び出してから) それぞれの関数のヘルプを御覧頂きたい.

20.1　因子分析

データ v2 はデータ v1 にノイズを加えたデータで，v4 と v3 , v6 と v5 も同様な関係となっている.

```
> v1 <- c(1,1,1,1,1,1,1,1,1,1,3,3,3,3,3,4,5,6); v2 <- c(1,2,1,1,1,1,2,1,2,1,3,4,3,3,3,4,6,5)
> v3 <- c(3,3,3,3,3,1,1,1,1,1,1,1,1,1,1,5,4,6); v4 <- c(3,3,4,3,3,1,1,2,1,1,1,1,2,1,1,5,6,4)
> v5 <- c(1,1,1,1,1,3,3,3,3,3,1,1,1,1,1,6,4,5); v6 <- c(1,1,1,2,1,3,3,3,4,3,1,1,1,2,1,6,5,4)
> m1 <- cbind(v1,v2,v3,v4,v5,v6)
> cor(m1)
          v1        v2        v3        v4        v5        v6
v1 1.0000000 0.9393083 0.5128866 0.4320310 0.4664948 0.4086076
v2 0.9393083 1.0000000 0.4124441 0.4084281 0.4363925 0.4326113
v3 0.5128866 0.4124441 1.0000000 0.8770750 0.5128866 0.4320310
v4 0.4320310 0.4084281 0.8770750 1.0000000 0.4320310 0.4323259
v5 0.4664948 0.4363925 0.5128866 0.4320310 1.0000000 0.9473451
v6 0.4086076 0.4326113 0.4320310 0.4323259 0.9473451 1.0000000
> factanal(m1, factors=3) # varimax is the default
Call:
factanal(x = m1, factors = 3)
Uniquenesses:
   v1    v2    v3    v4    v5    v6
0.005 0.101 0.005 0.224 0.084 0.005
Loadings:                                      # 固有ベクトルそのもの
   Factor1 Factor2 Factor3
v1 0.944   0.182   0.267
v2 0.905   0.235   0.159
v3 0.236   0.210   0.946
v4 0.180   0.242   0.828
v5 0.242   0.881   0.286
v6 0.193   0.959   0.196
```

```
               Factor1 Factor2 Factor3
SS loadings      1.893   1.886   1.797
Proportion Var   0.316   0.314   0.300
Cumulative Var   0.316   0.630   0.929

The degrees of freedom for the model is 0 and the fit was 0.4755
```

よって，v2 と v1，v4 と v3，v6 と v5 が上手く選別できていることが分かる．

20.2 正準相関分析

正準相関分析を行なう場合は関数 cancor() を用いる．

```
> library(mva)
> data(LifeCycleSavings)
> pop <- LifeCycleSavings[, 2:3]
> oec <- LifeCycleSavings[, -(2:3)]
> cancor(pop, oec)                               # 一般的な教科書で知られている出力とは異なる
$cor                                             # 変数群の固有値（正準相関係数）
[1] 0.8247966 0.3652762
$xcoef                                           # 変数 x の推定係数
              [,1]        [,2]
pop15 -0.009110856 -0.03622206
pop75  0.048647514 -0.26031158
$ycoef                                           # 変数 y の推定係数
             [,1]          [,2]          [,3]
sr   0.0084710221  3.337936e-02 -5.157130e-03
dpi  0.0001307398 -7.588232e-05  4.543705e-06
ddpi 0.0041706000 -1.226790e-02  5.188324e-02
$xcenter                                         # 変数 x を調節するのに使った値
  pop15   pop75
35.0896  2.2930
$ycenter                                         # 変数 y を調節するのに使った値
       sr       dpi      ddpi
   9.6710 1106.7584    3.7576
```

20.3 判別分析

線形を仮定した判別分析を行なう場合はパッケージ MASS の中の関数 lda() を用いる．関連のある関数として predict.lda() というものもある．

```
> library(MASS)
> data(iris3)
> Iris <- data.frame(rbind(iris3[,,1], iris3[,,2], iris3[,,3]),
+                     Sp = rep(c("s","c","v"), rep(50,3)))
> train <- sample(1:150, 75)
```

```
> table(Iris$Sp[train])      # your answer may differ
 c  s  v                     #  c  s  v
25 25 25                     # 22 23 30
> z <- lda(Sp ~ ., Iris, prior = c(1,1,1)/3, subset = train)
> predict(z, Iris[-train, ])$class
 [1] s s s s s s s s s s s s s s s s s s s s s s s s c c c c c c c c c c c c c c
[39] c c v c c c c c c c c c v v v v v v v v v v v v v v v v c v v v v v v v v
Levels: c s v
```

二次の判別分析を行なう場合はパッケージ MASS の中の関数 qda() を用いる．関連のある関数として predict.qda() というものがある．

```
> data(iris3)
> tr <- sample(1:50, 25)
> train <- rbind(iris3[tr,,1], iris3[tr,,2], iris3[tr,,3])
> test <- rbind(iris3[-tr,,1], iris3[-tr,,2], iris3[-tr,,3])
> cl <- factor(c(rep("s",25), rep("c",25), rep("v",25)))
> z <- qda(train, cl)
> predict(z,test)$class
 [1] s s s s s s s s s s s s s s s s s s s s s s s s c c c c c c c c c c c c c c
[39] c c c c c v c c c c c c v v v v v v v v v v v v v v v v v v c v v v v v v
Levels: c s v
```

20.4 主成分分析

主成分分析を行なう場合は関数 prcomp() を用いる．

```
> data(USArrests)
> prcomp(USArrests)
Standard deviations:
[1] 83.732400 14.212402  6.489426  2.482790
Rotation:
                PC1         PC2         PC3         PC4
Murder   -0.04170432 -0.04482166  0.07989066  0.99492173
Assault  -0.99522128 -0.05876003 -0.06756974 -0.03893830
UrbanPop -0.04633575  0.97685748 -0.20054629  0.05816914
Rape     -0.07515550  0.20071807  0.97408059 -0.07232502
> prcomp(USArrests, scale = TRUE)
Standard deviations:
[1] 1.5748783 0.9948694 0.5971291 0.4164494
Rotation:
               PC1        PC2        PC3         PC4
Murder   -0.5358995 -0.4181809  0.3412327 -0.64922780
Assault  -0.5831836 -0.1879856  0.2681484  0.74340748
UrbanPop -0.2781909  0.8728062  0.3780158 -0.13387773
Rape     -0.5434321  0.1673186 -0.8177779 -0.08902432
```

```
> plot(prcomp(USArrests))
> summary(prcomp(USArrests, scale = TRUE))
Importance of components:
                        PC1   PC2    PC3    PC4
Standard deviation     1.57 0.995 0.5971 0.4164
Proportion of Variance 0.62 0.247 0.0891 0.0434
Cumulative Proportion  0.62 0.868 0.9566 1.0000
```

20.5　クラスター分析

クラスター分析を行なう場合は関数 hclust() を用いる．

```
> data(USArrests)
> hc <- hclust(dist(USArrests), "ave")
> plot(hc)
> plot(hc, hang = -1)
> hc <- hclust(dist(USArrests)^2, "cen")
> memb <- cutree(hc, k = 10); cent <- NULL
> for(k in 1:10) cent <- rbind(cent, colMeans(USArrests[memb == k, , drop = FALSE]))
> hc1 <- hclust(dist(cent)^2, method = "cen" , members = table(memb))
> opar <- par(mfrow = c(1, 2))
> plot(hc, labels = FALSE, hang = -1, main = "Original Tree")
> plot(hc1, labels = FALSE, hang = -1, main = "Re-start from 10 clusters")
> par(opar)
```

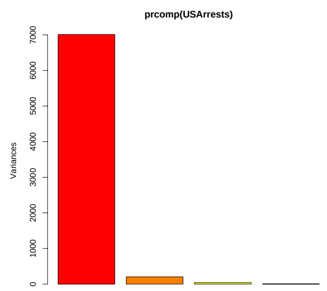


図 20.1　正準相関分析結果

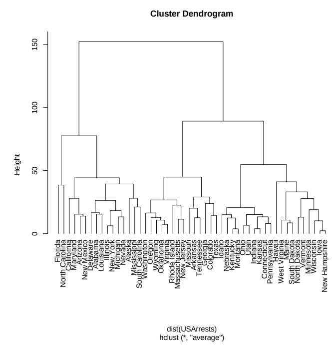


図 20.2　クラスター分析結果 (1)

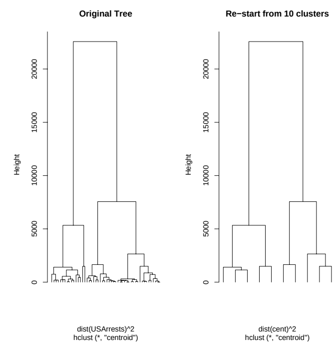


図 20.3　クラスター分析結果 (2)

k-means アルゴリズムによるクラスター分析を行う場合は以下のようにする．

```
> library(mva)
> x <- rbind(matrix(rnorm(100, sd = 0.3), ncol = 2),
+            matrix(rnorm(100, mean = 1, sd = 0.3), ncol = 2))  # 2D example
> cl <- kmeans(x, 2, 20)
> plot(x, col = cl$cluster)                                     # プロット結果は省略
> points(cl$centers, col = 1:2, pch = 8)
```

20.6　時系列解析

R には時系列解析のための関数が多数用意されている．詳しくは『R による統計解析の基礎』(中澤 港 著，ピアソン・エデュケージョン)，『THE R BOOK』 岡田 昌史 他 著 (九天社) を参照されたい．

関数	機能
ts()	時系列オブジェクトを生成する．as.ts()，is.ts() も用意されている．
ts.union()	時系列オブジェクトを合併する．
ts.intersection()	時系列オブジェクトの共通部分を求める．
window()	時系列オブジェクトの一部分を切りとる．
diff()	要素の差分を取る．
diffinv()	要素の差分をどんどん足していく．
filter()	要素の線形フィルタリングを行う．
lag()	時間軸を過去にずらす．
acf()	時系列オブジェクトの自己共分散と自己相関係数を求める．
aggregate()	データを部分集合に分割し（引数 nfrequency で指定)，それぞれの要約統計量を計算する．
ar()	時系列オブジェクトへの AR モデルの当てはめを行う．関連する関数として ar.burg()，ar.yw()，ar.mle()，ar.ols() がある．
arima()	時系列オブジェクトへの ARIMA モデルの当てはめを行う．関連する関数として arima.sim() がある．
ARMAacf()	ARMA モデルの自己相関係数を計算する．関連する関数として ARMAtoMA() がある．
Box.test()	時系列オブジェクトについて独立性の検定を行う．
ccf()	二つの一次元時系列間の相関係数共分散を求める．
decompose()	時系列を，移動平均により季節，傾向，不規則成分に分解する．似たような関数に stl() がある．
pacf()	部分自己相関係数 (partial autocorrelations) を計算する．
PP.test()	時系列オブジェクトが単位根をもつかどうかの検定を行う．
spectrum()	スペクトル密度関数を推定する．関連する関数として spec.ar() や spec.pgram()，spec.taper() がある．
cpgram()	累積ピリオドグラムをプロットする．
lag.plot()	時系列のラグプロットを行う．
monthplot()	時系列の季節成分をプロットする．
ts.plot()	複数の時系列を一つの画面にプロットする．

関数 ts.plot() は任意個の時系列ベクトルを与えてそれらを一度にプロットさせることが出来る[*1]．また，色や線の種類，点の記号などを時系列ベクトル毎に変更することも出来る．

```
> library(ts)
> data(UKLungDeaths)
> ts.plot(ldeaths, mdeaths, fdeaths,
+         gpars=list(xlab="year", ylab="deaths", lwd=c(1:3), col=c(2:4)))
```

[*1] plot() に時系列ベクトルを与える場合は，一つの時系列だけしか与えることが出来ないことに注意．

20.7　生存時間解析

急性骨髄白血病データ aml の内訳は

time：　「生存時間」または「打ち切りまでの時間」

status：　フラグ（０：打ち切り，１：イベント）

x：　群（Maintained：化学療法維持群，　Nonmaintained：非維持群）

となっている[*2]．カプラン・マイヤー法によるメディアン生存関数は関数 survfit() で求めることが出来，その結果をプロットすることでカプラン・マイヤープロットが得られる．

```
> library(survival)
要求されたパッケージ splines をロード中です
> MYDATA <- aml
> MYDATA$time2 <- Surv(MYDATA$time, MYDATA$status)   # time2 の打ち切りには + がつく
> ( result <- survfit(time2 ~ x, data=MYDATA) )      # メディアン生存関数
Call: survfit(formula = time2 ~ x, data = MYDATA)
                 n events median 0.95LCL 0.95UCL
x=Maintained    11      7     31      18     Inf
x=Nonmaintained 12     11     23       8     Inf
> summary(result)                                    # 詳しい要約を表示（結果は省略）
> plot(result, col=c(1,2))                           # カプラン・マイヤープロット
> legend(100, 0.8, c("Maintained","Nonmaintained"), col=c(1:5), lwd=1, bg="gray")
```

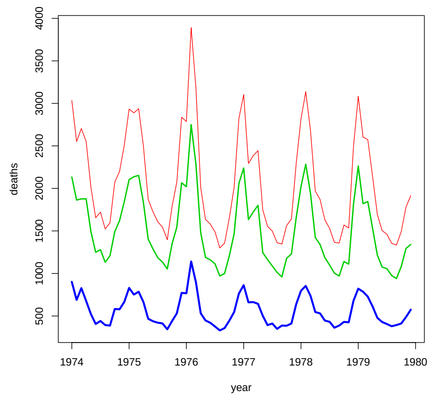


図 20.4　時系列プロット

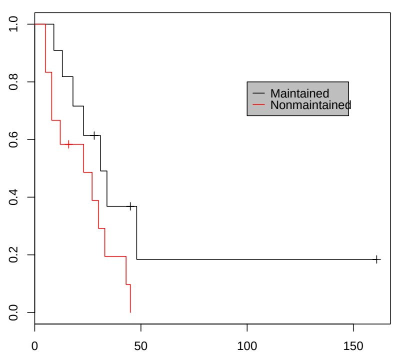


図 20.5　カプラン・マイヤープロット

ちなみに，ログランク検定は関数 survdiff() で，コックス回帰は関数 coxph() で実行できる[*3]．詳しくは『R による統計解析の基礎』(中澤 港 著，ピアソン・エデュケーション)，『THE R BOOK』 岡田 昌史 他 著 (九天社) を参照

[*2] 関数 Surv(観察開始時間, 観察終了時間, 打ち切りフラグ) で「0：右側打ち切り，1：イベント，2：左側打ち切り，3：区間打ち切り」という打ち切りフラグを生成することが出来る．

[*3] ベースライン（時間 0）で重症度が異なる場合，関数 strata() でベースライン値を層別因子として（共変量として）分析することが出来る．

```
> coxph(time2 ~ x + strata(DEMOG) , data=MYDATA) # 層によってベースラインハザードが異なると仮定
```

されたい．

20.8　ノンパラメトリック回帰

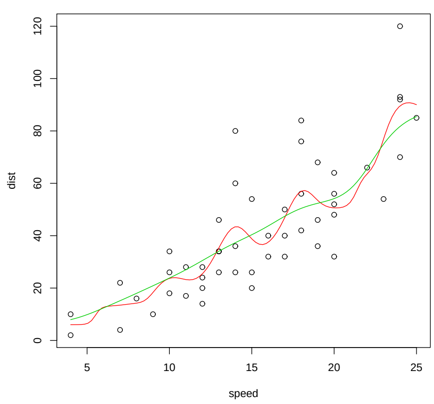
詳しくは『みんなのためのノンパラメトリック回帰 (上・下)』 (竹澤 邦夫 著，吉岡書店) を参照されたい．まず，ナダラヤ・ワトソン推定量を用いた平滑化は以下のようにすれば良い．

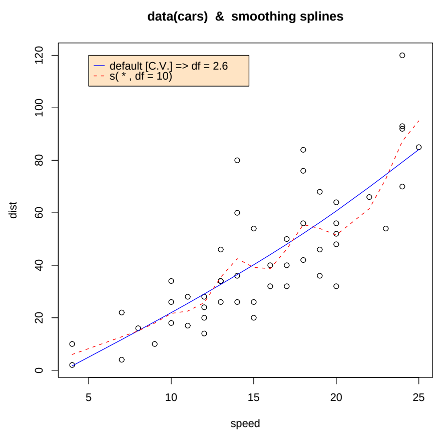
```
> library(modreg)
> data(cars)
> with(cars, {
+        plot(speed, dist)
+        lines(ksmooth(speed, dist, "normal", bandwidth=2), col=2)
+        lines(ksmooth(speed, dist, "normal", bandwidth=5), col=3)
+      })
```

平滑化スプラインによる平滑化を行う場合は以下のようにする．

```
> data(cars)
> attach(cars)
> plot(speed, dist, main = "data(cars)  &  smoothing splines")
> ( cars.spl <- smooth.spline(speed, dist) )
Call:
smooth.spline(x = speed, y = dist)

Smoothing Parameter  spar= 0.7801305  lambda= 0.1112206 (11 iterations)
Equivalent Degrees of Freedom (Df): 2.635278
Penalized Criterion: 4337.638
GCV: 244.1044
> lines(cars.spl, col = "blue")
> lines(smooth.spline(speed, dist, df=10), lty=2, col = "red")
> legend(5,120,c(paste("default [C.V.] => df =",round(cars.spl$df,1)),
+        "s( * , df = 10)"), col = c("blue","red"), lty = 1:2, bg='bisque')
```





引用文献・参考文献・引用サイト・参考サイト

[1] The R Project：http://www.r-project.org/
R の総本山です.

[2] 『S によるデータ解析』渋谷 政昭，柴田里程 著 (共立出版)

[3] 『S 言語 - データ解析とグラフィックスのためのプログラミング環境 (1) -』渋谷 政昭，柴田里程 訳 (共立出版)

[4] isac の S のリファレンスマニュアル：http://s.isac.co.jp/
(上の 3 つについて) S のコマンドの 9 割以上が R で使えますし，基本的な命令は (元が同じなので) 全く差異を感じることなく移植できます．よって，これらの書籍やホームページの内容は R を勉強する上で大いに参考になります.

[5] 間瀬先生の HP：http://www.is.titech.ac.jp/ mase/R.html
日本語の R の総本山とも云うべきサイトです．英語版マニュアルの日本語訳や，英語のヘルプの日本語訳などが置かれています．この頁を作成するに当たっては英語版マニュアルの日本語訳『 R 入門』『 R の言語仕様』などを参考にしました.

[6] R による統計処理 (青木先生の HP)：http://aoki2.si.gunma-u.ac.jp/R/
膨大な数の統計解析用プログラム，そこに付随する明快な解説，分かりやすい例，とにかく有益なサイトです．Mac 版 R のインストール方法なども載っています.

[7] なかまさんの HP：http://r.nakama.ne.jp/
日本語化パッチや，日本語化パッチ適用済のバイナリやソースが置かれています．感謝々々！！

[8] RjpWiki (岡田先生) ：http://www.okada.jp.org/RWiki/index.php?RjpWiki
RjpWiki はオープンソースの統計解析システム R に関する日本語による情報交換を目的とした Wiki です．ここでは Wiki とよばれるシステムを採用し，どなたでも自由にページを追加・編集できます.

[9] R・HTML マニュアル検索 (岡田さん)：http://www.okada.jp.org/cgi-bin/Rsearch/namazu.cgi/
R の html マニュアルを検索することが出来ます．これは非常に便利です！本当に便利です！

[10] 『R による統計解析の基礎』 中澤 港 著 (ピアソンエデュケーション)
R 人口が爆発的に増えるきっかけとなった中澤先生の本です．統計解析に関する詳しい解説と便利な R のソースで埋め尽くされています．しっかし，この中澤先生の知識量は何なのでしょうか！

[11] 『工学のためのデータサイエンス入門』 間瀬 茂・神保 雅一・鎌倉 稔成・金藤 浩司 著 (数理工学社)
間瀬先生が遂に R と統計に関する本を執筆． R ソースの解説と統計に関する解説が絶妙なバランスを保っています．非常に興味深いコラムやシミュレーションに関する章も見逃せない一冊です.

[12] 『The R Book』 岡田 昌史 他 著 (九天社)
遂に R のバイブルが登場！ OS 別実行ファイル，R の使用法の丁寧な解説，豪華執筆陣による R の凄まじいアプリケーションが 1 冊にまとまっています．商用ソフトならこれだけで数十万円する代物ですよ，これは！

[13] 『THE R Tips』 舟尾 暢男 著 (九天社)：この PDF の書籍版です．「 R 入門編」を追記しております.

[14] 『みんなのためのノンパラメトリック回帰 (上・下)』 竹澤 邦夫 著 (吉岡書店)
私がノンパラ密度推定を勉強する際にお世話になり，今は R-Tips を管理をお任せしているということでお世話になっている竹澤先生の著書．ノンパラ回帰・ウェーブレット・ヒストグラムの平滑化と密度推定を勉強しようという方にはお勧め．詳細な解説と膨大な R (と S-Plus) のソースが用意されています.

[15] 『フリーソフトウェア R による統計的品質管理入門』 荒木 孝治 編著 (日科技連)
この PDF では扱いませんでしたが，GUI 画面で R を操作することが出来るパッケージ『R commander』の詳しい解説が載っております．ちなみに，パッケージ『R commander』を日本語化されたのは荒木先生であります！

この PDF はこの頁で終了です．これまで (05/08/08) にたくさんの時間，ビールとおつまみを消費してきまして，今度こそ満足のいく内容になったと悦に入っております．普段の仕事を忘れ，ビールとつまみを口に運びながら作業をするのは至福でさえありました．一つ云えることは，以上の書籍・ウェブ頁・掲示板が無ければ，この PDF を作成することは出来なかったということです．これらの書籍の執筆者の方々，ウェブ頁の管理人様，RjpWiki の投稿者の方々に深くお礼を申し上げます．最後に，この PDF を管理して頂いている竹澤邦夫先生に心から感謝を申し上げます.