R 入門

R ノート: 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
Version 1.7.0 (2003-04-16)

**W. N. Venables, D. M. Smith
and the R Development Core Team**

日本語訳注：この R-intro の日本語訳は，英語原文と全く同じ条件の下で，自由に配布，利用，修正可能である．R の開発の早さから，こうした文章の日本語訳は常に "旧式化" していることをお断りしておく．最新バージョンの R 付属文章を適宜参照されたい．
この R-intro はその前身 R-notes の有志による邦訳を元に，間瀬茂 (東京工業大学) が翻訳したものを，第 1.7.0 版を元に小野肇 (TGO) の協力で改訂 (2003.4) したものである．なおこの邦訳を含む R の各種マニュアルの日本語訳のソースやその dvi, ps, html, pdf 版が入手可能[1] である．
R プロジェクト関連の文章の日本語化を目指して作られた有志のご厚意によるメイリングリスト[2] がある．
R-intro は GNU texinfo と呼ばれるマニュアル専用の TeX の方言で書かれており，TeX でコンパイル
[3] する．

ISBN 3-901167-55-2

[1] http://www.is.titech.ac.jp/~mase/R.html
[2] 詳細は http://epidemiology.md.tsukuba.ac.jp/~mokada/ml/R-jp.htmlを参照．このマニュアルを含む R に関する意見・質問はこのメイリングリストへどうぞ．
[3] 日本語版は，例えば日本語化 TeX の ascii ptex を用いるなら，まず "ptex R-intro-170.jp.texi"，次に索引作成のため "texindex R-intro-170.jp.*"，そしてもう一度 "ptex R-intro-170.jp.texi" と 3 段階でコンパイル．texinfo.tex が完全には日本語化されていないので，一部日本語化されない．

Table of Contents

序

このRのイントロダクションはBill VenablesとDaid M. Smith (Insightful Corporation) によって書かれたSとS-PLUS環境のオリジナル版ノートを元に書かれている．SとS-PLUSのプログラム機能での相違点を反映して若干の修正がされ，いくつかの内容は拡張されている．

このような改訂版を配布する許可を与え，影でRのサポートを与えてくれているBill Venablesに厚く感謝したい．

読者への注意

ほとんどのRの初心者はAppendix Aの入門セッションからスタートするだろう．こうすることによりRにおけるセッションのスタイルにある程度なれることができるだけでなく，より重要な点は実際にどんなことが出来るのかを直ちに体験することができるであろう．

主にグラフィックス機能を目当てにRを使うユーザが多いであろう．そうした場合には，グラフィックス機能を扱った章Chapter 12 [グラフィックス], page 62をいつでも必要な時に参照すればよく，その前のすべての章の内容を理解するまで待つ必要はない．

1 紹介と準備

1.1 R 環境

R はデータ操作，計算，グラフィックス表示といったソフトウェア機能の統合環境である．中でも次のような機能を持つ

- 効率的なデータの操作と蓄積機能，
- 配列，特に行列に対する一連の操作，
- データ解析における中間的な操作のための，豊富で一貫した，そして統合された道具，
- データの解析や表示を直接計算機で表示したり，印刷するためのグラフィック機能，そして
- 条件実行，ループ，ユーザ定義の再帰的関数，そして入出力機能を持つ，周到に開発された単純で効率的なプログラミング言語．(実際，システムに用意された関数のほとんどは，それ自身 S 言語によって書かれている．)

「環境」という言葉は，他のデータ解析ソフトにしばしば見られるような，次々と付け加えられた，極めて局限され融通の効かない道具の蓄積ではなく，それが十分に計画され一貫したシステムであるとことを特徴づけるために用いられている．

R は対話的なデータ解析の新たに発展中の手法に対する，まさに受け皿である．そのようなものとして，ダイナミックに変化しており，新しいリリースは，必ずしも以前のリリースと完全には上位互換とはなっていない．新しいリリースがもたらす新しい技術や手法を歓迎するユーザもいる一方で，古いコードがもはや使えないという事実に一層困惑するユーザもいるであろう．R はプログラミング言語として開発されているが，R で書かれた多くのプログラムは短命なものと考えててほしい．

1.2 関連するソフトウェアや文書

R は，ベル研究所において Rick Becker，John Chambers，そして Allan Wilks によって開発され，S-Plus システムのもとにもなっている S 言語を移植したものと見なすことができる．

S の発展は Jhon Chambers と共著者らによる 4 冊の本により特徴付けられる．R にとっての基本的文献は，Rick A. Becker，John M. Chambers， そして Allan R. Wilks によって書かれた『The New S Language: A Programming Environment for Data Analysis and Graphics』[1] である．S の 1991 年 8 月のリリース (S の第 3 版) における新しい機能は John M. Chambers と Trevor J. Hastie の編集による『Statistical Models in S』[2] に盛り込まれている．正確な文献は Appendix F [参考文献], page 92 を見よ．

そのほかにも S/S-Plus 用のドキュメントは一般的に使用できる．S の実装との違いを心に留めておくこと．マニュアル "R-FAQ, The R statistical system FAQ" 中の節 "R にはどんなドキュメントがあるの？"を参照．

[1] 訳注：邦訳『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 I，II』，渋谷政昭・柴田里程訳，共立出版 (1991)

[2] 訳注：邦訳『S と統計モデル —データ科学の新しい波—』，柴田里程訳，共立出版 (1994)

1.3 R と統計学

我々の R 環境への入門は「統計学」に付いて言及していなかったが，多くの人は R を統計システムとして使うであろう．我々はそれを多くの古典的，現代的な統計学手法が移植された環境と考えたい．これらのいくつかは基本的な R 環境に組み込まれているが，多くは「パッケージ」として提供されている．(現在のところ，違いは主に歴史的な偶然による．) R とともに提供されているパッケージは約 8 種類 (“標準” パッケージと呼ばれる) ある．もっと多くのパッケージが CRAN のインターネットサイト (`http://cran.r-project.org` 経由) で入手できる．

ほとんどの古典的統計学手法と最新の方法論の多くが R で使えるが，ユーザはそれを見つけ出すのに少々手間取ることを覚悟する必要があるだろう．

S(したがって R) と他の主要な統計システムの間には重要な哲学的違いがある．S では統計解析は普通何段階かのステップで行われ，途中結果はオブジェクトに保管される．したがって SAS や SPSS は回帰や判別分析に対し，おびただしい出力を与えるが，R は最小限の出力を与え，他の R 関数を用いた引き続く吟味のために，結果を当てはめオブジェクトとして保存する．

1.4 R とウィンドウシステム

R がもっとも便利に使えるのは，ウィンドウシステムが動くグラフィックスワークステーションにおいてであろう．この案内もこの機能が使えるユーザを対象にしている．説明のほとんどすべては R 環境のあらゆる移植に対して適用されるが，特に X-ウィンドウシステム上での R の用法にしばしば言及する．

多くのユーザは折にふれ，使用中の計算機の OS を直接操作する必要があるであろう．この案内では Unix マシンの OS との連携を中心に記述する．Windows で R を動かしているなら，適当な小さな修正が必要になるであろう．

R のカスタマイズ可能な特徴を最大限に活用するようにワークステーションを設定することは，少々面倒ではかも知れないが特に問題はなく，ここでは今後もあまり触れないことにする．困難を感じたときは，周囲のエキスパートの助けを求めるべきである．

1.5 R を対話的に使用する

R プログラムを使うと，命令の入力が期待されるときはプロンプトを表示する．既定ではプロンプトは ‘>’ になっている．Unix ではひょっとすると，シェルのプロンプトと同じかもしれず，したがって何も起こらないないように見えるかも知れない．しかしながら，後で見るように，必要なら R のプロンプトを別の物に変えるのは容易である．この文書では今後 Unix のシェルプロンプトは ‘$’ であると仮定する．

Unix で R を使う際，推奨される最初に行う手順は次の通りである：

1. まずこの問題に対し R で作業するデータファイルを置く，独立したサブディレクトリ (例えば ‘`work`’) を作る．これはこの特別な問題に対し R を使う際には，いつも作業ディレクトリになるであろう．

   ```
   $ mkdir work
   $ cd work
   ```

2. 次の命令で R を起動する

   ```
   $ R
   ```

3. ここで R の命令を入力できる (後を見よ) .
4. R を終了する命令は

```
> q()
```

ここで R セッションのすべてのデータを保存するかどうかを聞かれるであろう . 終了する前にデータを保存する , 保存しないで終了する , もしくは R セッションに戻るために , それぞれ *yes* , *no* または *cancel* (頭文字 1 つだけで十分) で答える . 保存されたデータは将来の R セッションで利用できる .

次回の R セッションは簡単である .

1. ‘work’ を作業ディレクトリにし , 前と同じようにプログラムを開始する :

```
$ cd work
$ R
```

2. R プログラムをを使い , セッションの最後で q() を用いて終了する .

Windows で R を使うときも , 従う手順は基本的に同じである . 作業ディレクトリとしてフォルダを作り , それを R のショートカット中の ‘Start In’ 欄にセットする . アイコンをダブルクリックし R を起動する .

1.6 入門セッション

先に進む前に , 計算機における R の雰囲気を得たいと思う読者は , Appendix A [入門セッション], page 76 にある入門的なセッションを実行してみることを強く勧める .

1.7 関数と特徴に関するヘルプを求める

R には UNIX の man 機能に似た組み込みのヘルプ機能が備わっている . 個々の名前のついた関数 , 例えば solve , について詳しいことが知りたいなら , 命令は

```
> help(solve)
```

である . もしくは

```
> ?solve
```

でも良い . 特殊文字で指定される機能が知りたいなら , 引数を 2 重引用符か 1 重引用符で囲み “文字列” にする必要がある :

```
> help("[[")
```

2 種類の引用符の各々は , "It's important" という例におけるように , 他方を通常文字に変換するのに使うことができる . 我々は便宜的に 2 重引用符を優先的に使うことにする .

R ではほとんどの場合ヘルプ文章は HTML 形式で得ることができ , 命令

```
> help.start()
```

を実行するとウエッブブラウザー (UNIX では netscape) を起動し , ヘルプ頁をハイパーリンクで閲覧できる . UNIX では引き続くヘルプの要求は HTML 形式のヘルプシステムに送られる .

help.start() によってロードされるページにある「Search Engine and Keywords」というリンクは , 高水準の概念の一覧表によって利用可能な関数から検索できるので特に有用である . 手早く答えを得るのに重宝するし , R が提供するべきものの広がりをよく理解できる .

help.search 命令によるヘルプの検索にはいろいろなやり方があるので ?help.search によって詳細と使用例を見てほしい．

ヘルプのトピックに関する例は次のようにして実行できる．

```
> example(topic)
```

R の Windows 版は他のオプションのヘルプシステムを持つ，詳細は命令

```
> ?help
```

を使用せよ．

1.8 R の命令，大・子文字の区別等．

技術的には R は非常に単純な構文を持つ「表現式言語」である．たいていの UNIX ベースのパッケージと同様に，「大文字と子文字を区別」し，したがって A と a は違った記号であり，違った変数として参照されるであろう．

R の名前に使えるシンボルのセットは，R が実行されているオペレーティングシステムと国に (技術的には使用している *locale* に) 依存する．ふつうは英数字すべて (国によってはアクセントのついた文字を含む) と '.' が使える[3] が，名前は数字で始まってはいけない．

初等的な命令は「表現式 (*expression*)」もしくは「付値 (*assignment*)」からなる．もしある表現式が命令として与えられると，それは評価され，表示され，そしてその返り値は失われる．付値も同様に，表現式を評価し，その値を変数に渡すが，結果は自動的には表示されない．

命令はセミコロン (';') か，新しい行で区切られる．複数の初等的命令は括弧 ('{' と '}') で括ることにより，一つの表現式に束ねることができる．注釈はほとんどどこにでも[4] 置くことができ，注釈マーク ('#') からその行の末尾までは注釈と見做される．

ある命令が行の最後で完結していなければ，R は別種のプロンプトを表示する，既定では

```
+
```

を 2 行目，そしてそれ以降の行に与え，命令が構文的に完成するまで入力を読み取り続ける．このプロンプトはユーザが変更できる．この文書では，この継続プロンプトは普通表示せず，単純な字下げで継続を指示することにする．

1.9 以前の命令の再呼び出しと修正

UNIX と Windows の多くのバージョンで，R は以前の命令の再呼び出しと再実行の機能を持っている．キーボードの垂直矢印キーを使って「命令履歴」を前後にスクロールすることができる．このようにして，いったんある命令が選ばれたら，命令内を水平矢印キーを用いてカーソル移動でき，文字を ⟨DEL⟩ キーで削除したり，他のキーで付け加えることもできる．より詳しくは後で説明される：Appendix C [行エディタ], page 88 を見よ．

再呼び出しや編集機能は大幅にカスタマイズできる．**readline** ライブラリに関するマニュアル項目を読めば，方法が分かるであろう．

あるいは Emacs テキストエディタが，R を用いた対話的な作業に関するより一般的な支援のメカニズム (ESS, *Emacs Speaks Statistics* を用いた) を提供してくれる．マニュアル "R-FAQ, The R statistical system FAQ" 中の節 "R と Emacs" を参照のこと．

[3] C のプログラマは '_' が使えないことと '.' が使えることに注意すること．後者は，R ではしばしば名前の中で語を区切るのに使われる．

[4] 文字列や関数定義の引数リスト中は除く

1.10 ファイルからの命令実行と，出力のファイルへの切替え

もし命令が現在の作業ディレクトリ ‘work’ にある外部ファイル，たとえば ‘commands.R’，に保存されているなら，それは R セッション中の任意の機会に，命令

```
> source("commands.R")
```

で実行できる．

Windows では **File** メニューの **Source** が同じように使える．関数 sink

```
> sink("record.lis")
```

は，それ以降の端末からのすべての出力を外部ファイル ‘record.lis’ に切替えるであろう．命令

```
> sink()
```

は，それをもう一度端末に戻す．

1.11 データの永続保存．オブジェクトの消去

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．それは変数，数の配列，文字列，関数，もしくはこれらの要素から作り上げられた，さらに一般的な構造であったりする．

一つの R のセッション中に，オブジェクトが作られ，名前を付けて保存される (このプロセスについては次のセッションで解説される)． R 命令

```
> objects()
```

(もしくは ls()) により，R に現在蓄積されているオブジェクトの名前を表示する現在蓄積されているオブジェクトの全体を作業スペース (*workspace*) と呼ぶ．

オブジェクトを削除するには，rm 関数が利用できる：

```
> rm(x, y, z, ink, junk, temp, foo, bar)
```

一つの R セッション中に作られた全てのオブジェクトは，将来の R セッションで使えるようにファイルに永続的に保存できる．各 R セッションを終了する際，現在利用できる全てのオブジェクトを保存する機会が与えられる．保存を選択すると，すべてのオブジェクトは現在のディレクトリの ‘.RData’ という名前のファイル[5] に書き込まれる．

後で R を起動したとき，このファイルから作業スペースを再読み込みする．と同時に関連する命令履歴も再読み込みされる．

R を用いた解析をするときは，別個の作業ディレクトリを使うべきである．ある解析において x や y といった名前のオブジェクトが作られることはよくあることである．このような名前は，単一の解析の文脈においてはしばしば意味を持つが，同一のディレクトリで複数の解析を行うならば，それが何を意味するかを判別するのは極めて困難になってしまう．

[5] このファイルの先頭の “ドット” は，それを UNIX の「隠しファイル」にする．

2 簡単な操作：数字とベクトル

2.1 ベクトルと付値

R は名前のついた「データ構造 (*data structures*)」を処理する．もっとも簡単なそうした構造は数値からなる「ベクトル (*vector*)」であり，順序づけられた数値の集まりからなる単一の対象である．5 つの数字 (たとえば 10.4, 5.6, 3.1, 6.4 そして 21.7) からなる x という名前のベクトルをつくるには，R の命令

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

を用いる．これは「関数 (function)」 c() を使った「付値 (*assignment*)」であり，この文脈では，任意の個数のベクトル「引数 (*arguments*)」を取ることができ，その返り値はその引数を端から端まで連結して得られるベクトル[1] である．

ある表現中に単独で現れた数値は，長さ 1 のベクトルと見なされる．

付値演算子 ‘<-’ は通常の ‘=’ で「ない」ことに注意しよう．等号記号は別の目的のためにとっておかれている．付値演算子は，真横に並ぶ 2 つの文字 ‘<’ (「より小さい」) と ‘-’ (「マイナス」) からなり，表現式の値を受け取るオブジェクトを「指して」いる．[2]

付値は assign() 関数を使うことによっても可能である．上記の代入操作と同様のことを行うには

```
> assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
```

とすればよい．通常の演算子 <- はこれの構文的な短縮形と考えることができる．

付値は同様に別向きで行うことも可能で，付値演算子の自明な変更を伴う．従って，同じことを

```
> c(10.4, 5.6, 3.1, 6.4, 21.7) -> x
```

としてもよい．もし表現が完全な命令として実行されるならば，その値が表示され，そして「失われる」[3]．だから，もし命令

```
> 1/x
```

を実行すると，5 つの値の逆数がターミナルに表示される (そして，もちろん x の値は変化しない)．

更に次の付値

```
> y <- c(x, 0, x)
```

は，x の 2 つのコピーの真ん中に零がある，11 個の項目からなるベクトル y を作るだろう．

2.2 ベクトル演算

ベクトルは算術表現中で使うことができ，その場合演算は要素毎に行われる．同じ表現中のベクトルは全て同じ長さである必要は無い．もし同じ長さでないならば，表現の返り値は，表現中の最も長い配列と同じ長さの配列になる．表現中のより短いベクトルは，それが最長のベクトルの長さに一致するまで (おそらく部分的に) 必要なだけ「リサイクル」される．特に定数は単純に繰り返される．したがって，上記の付値において，次の命令

---

[1] ベクトル引数以外の，list モードの引数のような場合，c() の行動はかなり異なる．Section 6.3.4 [任意のリストの登録], page 29 を参照せよ．

[2] 下線記号 ‘_’ は左向き付値演算子 ‘<-’ の代わりに使うことができるが，より可読性の低いコードにつながりやすく，勧められない．

[3] 実際は，他の付値が行われる前なら .Last.value 中に保存されている．

```
> v <- 2*x + y + 1
```

は，要素毎の和からなる長さ 11 のベクトル v を作るが，2*x が 2.2 回繰り返され，y は丁度一度だけ繰り返され，1 は 11 回繰り返される．

初等的な演算子は通常の +, -, *, /，そして冪乗 ^ である．更に普通の全ての算術関数を使うことができる．log, exp, sin, cos, tan, sqrt 等は全てそれらの通常の意味を持つ．max と min それぞれベクトル中の最も大きいものと最も小さいものを選択する．range() は長さ 2 のベクトル，つまり c(min(x), max(x)) を値に持つ関数である．length(x) は x の要素の数を返し，sum(x) は x の要素の総和を返し，prod(x) は全要素をかけた値を返す．

統計関数として，標本平均を計算する mean(x) があり，これは sum(x)/length(x) と同じである． var(x) は

```
sum((x-mean(x))^2)/(length(x)-1)
```

つまり標本分散を与える．もし var() の引数が $n\times\ p$ 行列なら，その値は列を独立な p-変量標本ベクトルとみなした $p\times\ p$ 標本共分散行列となる．

sort(x) は x と同じ長さの，要素を昇順に並べ変えた配列を返す．しかしながら，その他にももっと融通の効くソート機能がある (ソートのための置換を作る order() や sort.list() をみよ)．

max と min は，引数が複数のベクトルでも，引数中の最大と最小の値を選び出す．「並列化 (*parellel*)」最大・最小関数である pmax と pmin は，各入力ベクトルの位置にその最大 (最小) 値を置いた (長さがその引数の長さの最大値である) ベクトルを作り出す．

ほとんどの目的に取って，ユーザは数値ベクトル中の“数”が整数，実数，更には複素数かに関心が無いであろう．内部的な計算は倍精度実数か，もし入力データが複素数なら倍精度複素数で行われる．

複素数を使って作業するならば，虚数部を明示的に与える．したがって

```
sqrt(-17)
```

は NaN と警告を与えるが，しかし

```
sqrt(-17+0i)
```

は複素数として計算を行うであろう．

2.3 規則的な数列の生成

R はよく使われる数列を生成する幾つかの機能を持つ．例えば 1:30 はベクトル c(1,2, ...,29,30) である．

コロン演算子は一つの表現中で最も高い優先度を持つので，例えば 2*1:15 は c(2, 4, ..., 28, 30) というベクトルになる．n <- 10 と置いて，数列 1:n-1 と 1:(n-1) を比べてみよ．

構成 30:1 は降順の数列を作るのに使うことができる．

関数 seq() は数列を生成するもっと一般的な機能である．これは 5 個の引数を持つが，特定の呼出しでは，その一部分だけを指定すればよい．最初の 2 つの引数は．もし存在すれば．数列の最初と最後を指定し，もしこれだけが引数なら，結果はコロン演算子と同じになる．つまり seq(2,10) は 2:10 と同じベクトルになる．

他の多くの R の関数と同様に，seq() へのパラメータは名前付き形式で与えることができ，その場合にはそれらが現れる順序は勝手である．最初の 2 つのパラメータは from=*value* と to=*value* と名前を付けることができる．したがって，seq(1,30), seq(from=1, to=30), seq(to=30, from=1)

はすべて 1:30 と同じ結果になる．次の 2 つのパラメータは by=*value*, length=*value* という名前を付けることができ，それぞれ数列の増分と長さを指定する．もしどちらも与えられなければ，by=1 が既定として仮定される．

例えば

```
> seq(-5, 5, by=.2) -> s3
```

とすると s3 は ベクトル c(-5.0, -4.8, -4.6, ...,4.6, 4.8, 5.0) になる．同じく

```
> s4 <- seq(length=51, from=-5, by=.2)
```

とすると s4 も同じ配列になる．

5 番目のパラメータは along=*vector* という名前を付けることができ，もし使うのなら，このパラメータだけを指定しなければならない．これは 1, 2, ..., length(*vector*) というベクトルを作るか，もし空のベクトルを指定 (可能である) すると空の数列をつくる．関連した関数は rep() であり，さまざまな複雑なやり方でオブジェクトを複製するのに使うことができる．

最も簡単な例

```
> s5 <- rep(x, times=5)
```

は，x の 5 つのコピーを端から端へとつないだものを s5 にする．

2.4 論理ベクトル

数値ベクトルと同様に，R は論理量を扱うことができる．論理ベクトルの要素は TRUE，FALSE，NA(欠損値のこと，後述) という値を持つことができる．初めの 2 つはしばしばそれぞれ F と T と略される．ただし，F と T は予約語ではなく，既定で TRUE や FALSE に設定されている変数にすぎないので，ユーザが上書きすることが出来る．だから TRUE や FALSE を使用した方がよい．

論理ベクトルは「条件 (*conditions*)」 により生成される．例えば，

```
> temp <- x > 13
```

は temp を x と同じ長さで，x の対応する要素が条件を「満足しなければ」F を，「満足すれば」 T にしたベクトルにする

論理演算子には <, <=, >, >=，完全な一致を表す ==，不一致を表す!= がある．更に，もし c1 と c2 が論理表現なら c1 & c2 はそれらの論理積 (*“and”*)，c1 | c2 は論理和 (*“or”*)，そして !c1 は c1 の否定である．

論理ベクトルは通常の算術演算において使うことができ，そのときは数式ベクトルに「強制変換 (*coerced*)」され，F は 0 に， T は 1 になる．しかしながら，論理ベクトルとその強制変換された数式ベクトルが等価にならない場合がある．例は次の副節を参照のこと．

2.5 欠損値

ベクトルの要素が完全には知られていないことがある．統計的な意味で要素や値が「利用不能」や「欠損値」であるとき，ベクトル中のその位置に NA という特別な値を付値して保存しておくことができる．一般的には NA に対する処理は NA となる．この規則を設けた理由は単純で，もしある演算に対する指示が不完全ならば，結果を知ることができず，したがって利用不可能だからである．

is.na(x) 関数は x と同じ長さの論理ベクトルで，x 中の対応する要素の値が NA の時，そしてそのときのみ値 T を持つものを与える．

```
> z <- c(1:3,NA);  ind <- is.na(z)
```

NA は実際は値でなく，利用不可能な量に対する標識であるから，論理表現 x == NA と is.na(x) は全く別物であることを注意しよう．だから x == NA は x と同じ長さを持ち，その「全て」の値が NA であるベクトルである．論理表現自身が不完全でしたがって決定不能だからである．

数値計算によって生まれるもう一種の “欠損” 値がある．いわゆる「非数 (*Not a Number*)」である NaN 値である．例は

```
> 0/0
```

もしくは

```
> Inf - Inf
```

で，ともに意味のある結果が定義できないため NaN を生じる．

要約すると，is.na(xx) は NA と NaN の双方に対し TRUE となる．これらを区別するために is.nan(xx) は NaN に対してだけ TRUE となる．

2.6 文字ベクトル

例えばプロットのラベル等，R では文字や文字ベクトルを頻繁に使う．必要な場所で，それらは 2 重引用符文字で括られた文字列によって表現される．例えば，"x-values" や "New interatoin results" である．

文字列は 2 重引用符 (") か 1 重引用符 (') を使って入力されるが，出力するときは 2 重引用符である (引用符なしの場合もある)．文字列はエスケープ文字として \ を使った C スタイルのエスケープ

シーケンスを採用しているので，\ は \\ として入出力され，2 重引用符の中では " は \" とする．そのほか，改行 \n，タブ \t，バックスペース \b が有用である．

複数の文字ベクトルを c() 関数によって，1 つのベクトルに連結するすることができる．以下で多くの使用例が登場するであろう．

paste() 関数は任意の数の引数を取り，それらを 1 つの文字列に連結する．引数中に与えられた数値は自明の仕方で文字列に強制変換される．つまり，それらが印字された時にそうなるであろう形である．引数は，既定では 1 つの空白文字で区切られた形になるが，これは名前付きパラメータ sep=*string* で変更でき，区切り文字を *string* に変える．空文字 (区切り文字無し) も可能である．

たとえば，例

```
> labs <- paste(c("X","Y"), 1:10, sep="")
```

は labs を文字ベクトル

```
c("X1", "Y2", "X3", "Y4", "X5", "Y6", "X7", "Y8", "X9", "Y10")
```

にする．ここでも短いリストはリサイクルが行われるということを注意せよ；つまり，c("X","Y") は数列 1:10 にマッチするまで 5 回繰り返[4] される．

[4] paste(..., collapse=*ss*) は引数の間に文字列 *ss* を置くことにより単一の文字列にくっつけることができる．他にも文字操作用の多くの道具がある，sub 及び substring に対するヘルプを見よ．

2.7 添字ベクトル; データセットの一部分を選択する，変更する

ベクトルの要素の一部分は，ベクトル名に鍵括弧に入った「添字ベクトル (*index vector*)」をあてがうことにより選択することができる．より一般に，結果がベクトルとして評価される任意の表現式は，表現式の直後に鍵括弧に入った添字ベクトルを加えることにより，同じようにその要素の部分集合を得ることができる．

このような添字ベクトルは，4 つの異なったタイプのいずれでも良い．

1. 「論理ベクトル」．この場合，添字ベクトルは，要素が選び出されるベクトルと同じ長さを持つ必要がある．添字ベクトル中の TRUE に対応する値が選択され，FALSE に対応するものは無視される．例えば，

   ```
   > y <- x[!is.na(x)]
   ```

   は，x の欠損値でない値を，同じ順序に並べたオブジェクト y を作る (または y を作り変える)．もし x が欠損値を持てば，y は x よりも短くなることを注意しよう．同様に

   ```
   > (x+1)[(!is.na(x)) & x>0] -> z
   ```

   は，オブジェクト z をつくり，それに，ベクトル x+1 の対応する値が欠損値でなく正の値持つようなものを，要素として置く．

2. 「正の整数値ベクトル」．この場合，添字ベクトル中の値は集合{$1, 2, \ldots,$ length(x)} 中になければならない．対応するベクトルの要素が選び出され，結果中に「その順序で」まとめられる．添字ベクトルは任意の長さで良く，その結果は添字ベクトルと同じ長さとなる．例えば x[6] は x の 6 番目の成分であり

   ```
   > x[1:10]
   ```

   は x の最初の 10 個の要素が選び出す (length(x) が 10 未満で無いことを仮定すれば)．同様に

   ```
   > c("x","y")[rep(c(1,2,2,1), times=4)]
   ```

   は，4 回繰り返された "x","y","y","x" からなる，長さ 16 の文字ベクトルを作る (あまりしないことだろうが)．

3. 「負の整数値ベクトル」．このような添字ベクトルは，選択するよりも「除外されるべき」値を特定する，だから

   ```
   > y <- x[-(1:5)]
   ```

   は，x の最初の 5 つの要素を除いた残り全てからなる y を与える．

4.

   「文字列ベクトル」．これは，オブジェクトがその要素を特定するための名前属性を持つ場合のみに使われる．名前ベクトルのサブベクトルを，2 つ上の項「正の整数値ベクトル」と同様に使うことができる．

   ```
   > fruit <- c(5, 10, 1, 20)
   > names(fruit) <- c("orange", "banana", "apple", "peach")
   > lunch <- fruit[c("apple","orange")]
   ```

   文字と数字を組み合わせた「名前 (*names*)」は，しばしば「数字の添字」よりも覚えやすいという利点がある．あとで分かるように，このオプションはデータフレームとの関連でとりわけ役にたつ．

添字付き表現式はまた，1 つの付値の代入される側に現れることができ，そのときは付値演算は「ベクトルのそうした要素に対してのみ」実行される．表現式は vector[*index_vector*] の形を持たねばならない．ベクトル名の位置に勝手な表現を置くことは，この場合あまり意味がないからである．

付値されるベクトルは，添字ベクトルの長さと合致しなければならず，論理値添字ベクトルの場合は，添字操作されるベクトルと同じ長さでなければならない．

例えば

```
> x[is.na(x)] <- 0
```

は，x 中の欠損値を零に置き換え，そして

```
> y[y < 0] <- -y[y < 0]
```

は次と同じ効果を持つ

```
> y <- abs(y)
```

2.8 他の型のオブジェクト

ベクトルは R のもっとも重要なオブジェクトの型であるが，後の節でより形式的に登場する別の型もいくつかある．

- 「行列 (*matrices*)」もしくはより一般に「配列 (*arrays*)」はベクトルの多次元化である．実際，それらは 2 つ以上の添字で添字付けられることができ，特殊な方法で出力することができる，Chapter 5 [配列と行列], page 18 を見よ．
- 「因子 (*factors*)」はカテゴリ化されたデータを扱う簡潔な方法を与える，Chapter 4 [因子], page 16 を見よ．
- 「リスト (*lists*)」はベクトルの一般形で，その要素は同じ型である必要は無く，しばしばそれ自身ベクトルやリストであったりする．リストは統計計算の結果を返す便利な方法を与える．Section 6.1 [リスト], page 26 を見よ．
- 「データフレーム (*data frames*)」は行列に似た構造で，異なった型の列を持つことができる．データフレームとは各列毎に観測ユニット，但し (可能性として) 同時に数値かつカテゴリ化された変量を持つ，からなる ‘データ行列’ と考えると良い．多くの実験はデータフレームとしてもっとも良く記述できる；処理はカテゴリ化され，反応は数値で与えられる．Section 6.3 [データフレーム], page 27 を見よ．
- 「関数 (*functions*)」はそれ自身 R のオブジェクトで，プロジェクトの作業スペースに保存できる．これは R を拡張する単純で便利な方法を与える．Chapter 10 [自分自身の関数を書く], page 42 を見よ．

3 オブジェクト，そのモードと属性

3.1 本質的属性：モードと長さ

R が処理する対象は，技術的には「オブジェクト (*objects*)」として知られているものである．例えば，数値 (実数) ベクトルや複素数ベクトル，論理値ベクトルや文字列ベクトルである．これらは「アトミック (atomic) な」構造として知られている．なぜなら，それらが全て同じ型，もしくは「モード (*mode*)」，つまりそれぞれ「数値 (*numeric*)[1]」,「複素数 *complex*()」,「論理値 (*logical*)」，そして「文字 (*character*)」，からなるからである．

ベクトルは必ず「全て同じモードからなる」値を持たなければならない．だから，与えられたどのベクトルも，曖昧さ無しに，「論理」,「数値」,「複素数」もしくは「文字」のどれかでなければならない．唯一の些細な例外が，利用できない量を示す `NA` として言及された特別な「値」である．ベクトルは空であっても，モードを持つことを注意しよう．例えば，空の文字ベクトルは `character(0)` として，空の数値ベクトルは `numeric(0)` として言及できる．

R は「リスト (*lists*)」と呼ばれるオブジェクトも処理でき，それらは「リスト (*list*)」というモードである．これらは，それぞれが任意のモードを持つことができるオブジェクトの，順序づけられた列である．「リスト (*lists*)」は，原始的ではなく，「再帰的 (recursive) な」構造を持つ．なぜなら，その構成要素がそれ自身リストになり得るからである．

他の再帰的な構造として「関数 (*function*)」 と「表現式 (*expression*)」がある．「関数」は R システムの一部分をなすものと，ユーザが書いた同様な関数とがあり，後のほうである程度詳しく議論されるであろう．オブジェクトとしての表現式は R のより進んだ部分であり，この案内では，R におけるモデリングとともに用いられる「公式 (*formulae*)」を議論する際に間接的にふれる以外には，議論されないであろう．

オブジェクトの「モード (*mode*)」とは，その基本的構成物の基本的な型を意味する．これはオブジェクトの「性質 (property)」の特殊な例である．すべてのオブジェクトにも備わっているもう一つの特質はその「長さ (*length*)」である．関数 `mode(`*object*`)` と `length(`*object*`)` は，任意の既定義構造のモードと長さ

[2] を見出すのに使うことができる．

オブジェクトのその他の性質は普通 `attributes(`*object*`)` によって得ることができる，Section 3.4 [属性を得る・変える], page 15 を見よ．このために，*mode* と *length* はまたオブジェクトの「本質的な属性 (intrinsic attributes)」とも呼ばれる．

例えば，もし `z` が長さ 100 の複素数ベクトルなら，`mode(z)` は文字列 `"complex"` であり，`length(z)` は 100 である．

R は，それが意味あると思われるあらゆるところでモードの変更を行うことができる (いくつかはそうとは思えないような場合にもである)．例えば

```
> z <- 0:9
```

において

[1] 「数値」モードは実際には 2 つの異なったモード．つまり「整数 (*integer*)」と「倍精度実数 (*double*)」の混ざったものである．

[2] `length(`*object*`)` が，本質的な役に立つ情報を常に含んでいるとは限らないことに注意．たとえば *object* が関数である場合など．

```
> digits <- as.character(z)
```

おくと，digits は文字ベクトル c("0", "1", "2",..., "9") になる．もう一段の「強制変換 (*coercion*)」，つまりモードの変更，は再び数値ベクトルをつくり出す．

```
> d <- as.integer(digits)
> d <- as.integer(digits)
```

今や d と z はおなじもの[3] になる．as.*something*() の形式の，あるモードから他のモードへの強制変換，そしてオブジェクトにそれがまだ所有していないモードを与える，たくさんの関数がある．読者はそれらに慣れるために他のヘルプ文章を参考にすべきである．

3.2 オブジェクトの長さを変える

「空 (empty)」のオブジェクトもモードを持っているかも知れない．

```
> e <- numeric()
```

は e を数値モードの空のベクトルにする．同様に character() は空の文字ベクトルとなる，等々．いったん任意の長さのオブジェクトが作られると，新しい要素を，単に以前の範囲の外部の添字値を与えることにより，追加することができる．だから

```
> e[3] <- 17
```

は e を長さ 3 のベクトル (この時点で，最初の 2 つの値はともに NA) にする．これは，もし追加される要素達のモードが，最初のオブジェクトのモードと一致するならば，全く同じように任意の構造に適用できる．

この自動的なオブジェクトの長さの調整は，例えば入力のための scan() 関数において，しばしば使われる．(Section 7.2 [scan() 関数], page 31 を見よ．)

逆に，オブジェクトの長さを切り詰めるためには，単なる付値が必要になるだけである．したがって，もし alpha が長さ 10 のオブジェクトなら，

```
> alpha <- alpha[2 * 1:5]
```

により，それは最初に偶数の添字を持っていた要素だけからなる，長さ 5 のオブジェクトになる．もちろん，古い添字は保存されない．

3.3 属性を得る・変える

関数 attributes(*object*) は，そのオブジェクトにたいし現在定義されているすべての非本質的属性のリストを与える．関数 attr(*object*, *name*) は特定の属性を選ぶのに使うことができる．これらの関数が使われることは稀で，例外は，たとえばある R のオブジェクトにそれが作られた日時や演算子を関連づけるといった特別な目的のために，ある新しい属性が作られた場合である．しかし，この概念は極めて重要である．

属性は R のオブジェクトシステムの統合された一部分であり，それらを付加したり除去する際には注意しなければならない．

属性関数をある付値の左辺に用いた場合，それは オブジェクト に新しい属性を関連づけたり，すでに存在している属性を変更することができる．例えば

```
> attr(z,"dim") <- c(10,10)
```

は，R が z を 10× 10 行列であるかのように扱うことを可能にする．

[3] 一般に，数値から文字への強制変換と逆への強制変換は，文字表現における丸め誤差により，正確には同じものにはならない．

3.4 属性を得る・変える

オブジェクトの「クラス (*class*)」として知られる特別な属性は R におけるオブジェクト指向スタイルのプログラミングを可能にする．

例えば，あるオブジェクトがクラス `"data.frame"` を持てば，それはある仕方で表示され，`plot()` 関数はそれをある仕方でグラフィカルに表示し，そして `summary()` 等の他のいわゆる「総称的 (*generic* な)」関数は，その引数に対しそのクラスを意識した仕方で動作する．

クラスの効果を一時的に取り去るには，関数 `unclass()` を使う．例えば，もし `winter` がクラス `"data.frame"` を持てば，

```
> winter
```

はそれをデータフレーム形式，行列に似た形，で出力するが，一方で

```
> unclass(winter)
```

はそれを通常のリストとして出力する．この機能はかなり特別の状況でしか必要でないであろうが，これが必要になる一つの状況は，学習中にクラスと総称的関数というアイデアに思い悩んだ時であろう．

総称的関数とクラスに付いては更に Section 10.9 [オブジェクト指向], page 49 で議論されるが，簡単にだけである．

4 順序付き因子と順序無し因子

「因子 (*factor*)」とは，同じ長さを持つ別のベクトルの要素の離散的分類 (グループ化) を指示するベクトルオブジェクトである．R は「順序が付いた (*ordered*)」因子と「順序の無い (*unordered*)」因子の双方を扱うことができる．因子の真の用途はモデル公式 (Section 11.1.1 [コントラスト], page 52 を見よ) に対してであるが，ここでは一つの例を見よう．

4.1 特別な例

例えば，オーストラリアの全ての州と準州[1] からの 30 人の会計士の標本があるとし，それらの個々の本拠地の所在州が，州名の省略形の文字列によって次のように指定されているとする．

```
> state <- c("tas", "sa",  "qld", "nsw", "nsw", "nt",  "wa",  "wa",
             "qld", "vic", "nsw", "vic", "qld", "qld", "sa",  "tas",
             "sa",  "nt",  "wa",  "vic", "qld", "nsw", "nsw", "wa",
             "sa",  "act", "nsw", "vic", "vic", "act")
```

文字ベクトルの場合，「ソート」とはアルファベット順にソートすることを意味することを注意して欲しい．

「因子 (*factor*)」は同様に `factor()` 関数によって作ることができる．

```
> statef <- factor(state)
```

`print()` 関数は因子を他のオブジェクトとは少々違った仕方で扱う：

```
> statef
 [1] tas sa  qld nsw nsw nt  wa  wa  qld vic nsw vic qld qld sa
[16] tas sa  nt  wa  vic qld nsw nsw wa  sa  act nsw vic vic act
Levels:  act nsw nt qld sa tas vic wa
```

因子の水準を知るには，`levels()` 関数を使うことができる．

```
> levels(statef)
[1] "act" "nsw" "nt"  "qld" "sa"  "tas" "vic" "wa"
```

4.2 関数 `tapply()` と不揃い配列

直前の例を続け，同じ会計士の収入が (適当な大きな貨幣単位で) 別のベクトルに与えられているとしよう．

```
> incomes <- c(60, 49, 40, 61, 64, 60, 59, 54, 62, 69, 70, 42, 56,
               61, 61, 61, 58, 51, 48, 65, 49, 49, 41, 48, 52, 46,
               59, 46, 58, 43)
```

各州毎の収入の標本平均収入を計算するには，ここで特別な関数 `apply()` を使うことができる：

```
> incmeans <- tapply(incomes, statef, mean)
```

結果は水準をラベルに持つ要素からなる平均のベクトルである．

---

[1] 外国の読者は，オーストラリアには八つの州と準州，つまり the Australian Capital Territory, New South Wales, the Northern Territory, Queensland, South Australia, Tasmania, Victoria そして Western Australia があることを注意して欲しい．

```
   act    nsw     nt    qld     sa    tas    vic     wa
 44.500 57.333 55.500 53.600 55.000 60.500 56.000 52.250
```

関数 tapply() はある関数 (ここでは mean()) を，最初の引数 (ここでは incomes) の成分の，2 番目の引数 (ここでは statef

[2]) の水準で定義される各グループに，あたかもそれらが別個のベクトル構造であるかのように，適用するのに使われる．結果は，因子の水準属性と同じ長さの，結果を含む構造である．詳細に付いてはヘルプ文章を参照されたい．

更に州ごとの平均収入に対する標準誤差を計算する必要があるとする．このためには，与えられた任意のベクトルに対し，標準誤差を計算する R の関数を書く必要がある．標本分散を求める組み込み関数 var() があるので，この関数はとても簡単に一行で書け，次のような付値になる：

```
> stderr <- function(x) sqrt(var(x)/length(x))
```

(関数の書き方に付いては後の Chapter 10 [自分自身の関数を書く], page 42 で考えられる．) この付値を行えば，標準誤差は次のように計算される．

```
> incster <- tapply(incomes, statef, stderr)
```

そして計算された値は次のようになる．

```
> incster
act    nsw  nt    qld     sa tas   vic     wa
1.5 4.3102 4.5 4.1061 2.7386 0.5 5.244 2.6575
```

演習問題として，州毎の平均税収に対する 95% 信頼区間を計算してみることができるだろう．このためには，標本サイズを見出すため，再び apply() 関数を length() 関数とともに使用し，適当な t-分布のパーセント点を見出すために，関数 qt() を使用することができるだろう．

tapply() 関数は，複数のカテゴリによって分類されたベクトルの，より複雑な添字の操作を処理できる．例えば，税理士を州と性別によって分割したいとする．この簡単な例では，しかしながら，何が起こるかは次のように考えることができる．ベクトル中の値は，カテゴリ中の異なった項目に対応するグループに選別される．次に，これらのグループのそれぞれに対して個別に関数が適用される．結果の値は，関数値からなるベクトルであり，カテゴリの水準属性によりラベルが付けられている．

ベクトルとラベル因子の組合せは，サブクラスのサイズが不規則になる 可能性があるため，しばしば「不揃い配列 (*ragged array*)」と呼ばれるものの例になる．もしサブクラスのサイズが全て同じならば，次の節で見るように，添字操作は暗黙のうちに，そしてより効率的に行うことができる．

4.3 順序が付いた因子

因子の水準はアルファベット順に保管されている．もしくは，明示的に指示された場合は factor に指定された順序で保管される．

水準は我々が記録しておきたい，そしてそれを利用して統計解析を行いたい自然な順序を持つことがある．関数 ordered() はそうした順序の付いた因子を作り出すが，それ以外は factor と同じ機能を持つ．多くの場合に，順序付きと順序無しの因子の唯一の違いは，前者が水準の順序を示して表示されることにあるが，線形モデルの当てはめに於けるそれらから作り出されるコントラストは異なったものになる．

---

[2] tapply() はこの場合，その第 2 引数が因子でなくても使えることを注意しよう，例えば ‘tapply(incomes,state)’．引数は必要なときは因子に (as.factor() を用いて) 強制変換されるからである．こうしたことをおこなう関数は他には極めて少ない．

5 配列と行列

5.1 配列

配列とは何重にも添字付けられたデータ項目 (例えば数字) の集まりと考えることができる．R は配列，特別な場合として行列，を簡単に作ったり操作する機能を持つ．

次元ベクトルとは正の整数からなるベクトルである．もしその長さが k ならば配列は k-次元である，つまり行列は 2-次元配列になる．次元ベクトル中の値は k 個の添字の各々の上限を与える．下限は常に 1 である．

R ではベクトルは，もしそれが次元ベクトルをその *dim* 属性として持つ場合にのみ配列として用いることができる．例えば，`z` が 1500 個の要素からなるベクトルとしよう．付値

```
> dim(z) <- c(3,5,100)
```

はそれに *dim* 属性を与え，$3 \times 5 \times 100$ 配列として扱うことを許すようになる．

より単純で自然な代入のために `matrix()` や `array()` といった他の関数を用いることができる，Section 5.4 [array() 関数], page 20 を参照せよ．

データベクトル中の値は FORTRAN で使われるのと同じ順番で配列中の値に対応する．つまり，最初の添字が最も早く変わり，最後の添字が最もゆっくり変わるという列主導の順序である．

例えば配列 `a` の次元ベクトルが `c(3,4,2)` ならば，`a` の中には $3 \times 4 \times 2 = 24$ 個の項目があり，データベクトルはそれらを `a[1,1,1]`，`a[2,1,1]`，...，`a[2,4,2]`，`a[3,4,2]` の順序で保持している．

5.2 配列の添字．配列のサブセクション

配列中の個々の要素は上の例のように，配列の名前に続いて鍵括弧にはさまれ，コンマで分離された添字の列で参照することができる．

もっと一般に，配列の一部分は添字の代わりに「添字ベクトル」の列を与えることにより指定することができる．しかしながら，「もしある添字位置に空の添字ベクトルを与えると，その位置に可能なすべての添字が指示されたことになる」．

前の例を再び用いると，`a[2,,]` は次元ベクトル `c(4,2)` を持つ 4×2 配列で，値

```
c(a[2,1,1], a[2,2,1], a[2,3,1], a[2,4,1],
  a[2,1,2], a[2,2,2], a[2,3,2], a[2,4,2])
```

をこの順で持つデータベクトルである．`a[,,]` は配列全体を表すことになり，添字を省略し単に `a` と書くのと同じことになる．

任意の配列，例えば `Z`，の次元ベクトルは `dim(Z)` で明確に参照できる (付値式の両辺で用いることができる)．

また，配列名が「単一の添字もしくは添字ベクトル」とともに用いられると，データベクトルのそれに対応する値だけが用いられ，この場合次元ベクトルは無視される．しかしながら，次に示すように，単一の添字がベクトルでなく，それ自身配列ならばこの限りでは無い．

5.3 添字の配列

任意添字位置の添字ベクトルと同様に，配列の添字として，単一の「添字配列」を用いることにより，配列中の不規則な一部分に値を代入したり，不規則な一部分をベクトルとして取り出すことが出来る．

行列を例にとると手順がわかりやすい．2 重添字配列の場合，2 つの列と任意の数の行を持つ添字行列を考えることが出来る．この添字行列の項目は，2 重添字配列に対する行・列添字である．例えば 4×5 配列 X があり，次の操作をしたいとする：

- 要素 X[1,3], X[2,2] と X[3,1] をベクトルとして抽出し，
- 代わりに X 中の対応する値を 0 に置き換えたい．

この場合次の例にに示されたように，3×2 の添字配列が必要になる．

```
> x <- array(1:20,dim=c(4,5))   # 4× 5 の配列を生成
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
> i <- array(c(1:3,3:1),dim=c(3,2))
> i                             # i は 3× 2 の添字配列
     [,1] [,2]
[1,]    1    3
[2,]    2    2
[3,]    3    1
> x[i]                          # 対応する要素を抽出
[1] 9 6 3
> x[i] <- 0                     # これらの要素を 0 で置き換える
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    0   13   17
[2,]    2    0   10   14   18
[3,]    0    7   11   15   19
[4,]    4    8   12   16   20
>
```

より自明でない例として，2 つの因子 blocks (b 水準) と varieties (v 水準) を持つブロック実験計画用の (非既約な) 計画行列を作成したいとする．さらに，実験には n 回の繰り返しがあるとする．次のように進めれば良い．

```
> Xb <- matrix(0, n, b)
> Xv <- matrix(0, n, v)
> ib <- cbind(1:n, blocks)
> iv <- cbind(1:n, varieties)
> Xb[ib] <- 1
> Xv[iv] <- 1
> X <- cbind(Xb, Xv)
```

さらに一致行列，例えば N，を作成するには

```
> N <- crossprod(Xb, Xv)
```

とすることが考えられる．しかしながらこの行列を生成するより簡単な方法は `table()` 関数を用い

```
> N <- table(blocks, varieties)
```

とすることである．

5.4 array() 関数

ベクトル構造に `dim` 属性を与えるのと同様に，配列はベクトルから `array` 関数を用いて次のように生成できる．

```
> Z <- array(data_vector, dim_vector)
```

例えば，ベクトル `h` が 24 かそれ未満の要素を含むならば，命令

```
> Z <- array(h, dim=c(3,4,2))
```

は `h` を用いて $3 \times 4 \times 2$ の配列 `Z` を作り出すであろう．もし `h` のサイズがちょうど 24 ならば結果は次と同じになる．

```
> dim(Z) <- c(3,4,2)
```

しかしながら，もし `h` のサイズが 24 未満なら，サイズが 24 になるように，`h` の値が最初からリサイクルされる (Section 5.4.1 [リサイクル規則], page 20 を見よ)．極端であるがよくある例として

```
> Z <- array(0, c(3,4,2))
```

は `Z` をすべて零からなる配列とする．

この時点で `dim(Z)` は次元ベクトル `c(3,4,2)` を表し，`Z[1:24]` は `h` にあったのと同様なデータベクトルを表す．空の添字を持つ `Z[]` もしくは添字無しの `Z` は配列としての全体を表す．

配列は数値演算表現中で用いることが出来，結果はデータベクトルの要素毎に演算を行なった結果の配列となる．被演算項は普通同じ `dim` 属性を持つ必要があり，これが結果の配列の `dim` 属性になる．したがって `A`, `B`, `C` が同じような配列なら

```
> D <- 2*A*B + C + 1
```

は同じ `dim` 属性を持ち，要素毎に行なわれる自明な演算結果からなるデータベクトルを持つ配列 `D` を作る．しかしながら，配列とベクトルが交じりあった演算に関する正確な規則はもう少し慎重に考える必要がある．

5.4.1 異種類のベクトル，配列に対する演算．リサイクル規則

配列とベクトルが交じりあった演算における，各要素が被る正確な規則は少々気まぐれであり，参考になる文献を見つけることが困難である．経験から著者は次の指針が頼りになることがわかった．

- 表現は左から右に読み取られる．
- 前後の被演算項に比べ，要素数が不足する短いベクトル被演算項は，必要な要素数になるまでリサイクルされる．
- 短いベクトルと配列「のみ」が関係する限り，配列はすべて同じ `dim` 属性を持つ必要がある．さもなければエラーとなる．
- 行列や配列被演算項より長いベクトル被演算項はエラーを引き起こす．
- もし演算に配列が含まれ，エラーやベクトルへの強制変換が行なわれなければ，結果は配列被演算項の共通の `dim` 属性を持つ配列となる．

5.5 2 つの配列の外積

配列に対する重要な演算が「外積 (outer product)」である．a と b が 2 つの数値からなる配列とすると，それらの外積は，次元ベクトルが a と b の次元ベクトルの連結になり (順序が肝要)，データベクトルは a と b の要素のすべての可能な積の全体となる．外積は特殊な演算子 \%o\% で表される．

```
> ab <- a %o% b
```

もう 1 つの表現は

```
> ab <- outer(a, b, "*")
```

積演算は 2 つの変数を取るすべての関数に置換えることができる．例えば，もし関数 $f(x$ f(x, y) = cos(y)/(1 + x^2),

を，ふたつの R のベクトル x と y が x と y-座標になるような 2 次元格子上で評価したければ次のようにすれば良い：

```
> f <- function(x, y) cos(y)/(1 + x^2)
> z <- outer(x, y, f)
```

特に 2 つの普通のベクトルの外積は 2 重に添字を持つ配列 (ランクが最大でも 1 の行列) になる．外積は当然非可換な演算になることを注意しよう．ユーザ定義の R の関数の定義方法についてはさらに Chapter 10 [自分自身の関数を書く], page 42 で議論されるであろう．

1 つの例：2× 2 数値行列の行列式

人工的だが気のきいた例として 2× 2 行列 $[a, b; c, d]$ の行列式を考えよう．要素はすべて非負の整数 $0, 1, \ldots, 9$ を取るとする．

問題はこの形のすべての可能な行列に対する行列式 $ad - bc$ を見い出し，その各値が high density プロットとして現れるような頻度を表現することである．これは各数値が独立かつ一様に選ばれた時の行列式の確率分布を求めることに相当する．

これを手際良くやる 1 つの方法は outer 関数を 2 回使うことである：

```
> d <- outer(0:9, 0:9)
> fr <- table(outer(d, d, "-"))
> plot(as.numeric(names(fr)), fr, type="h",
       xlab="Determinant", ylab="Frequency")
```

行列式の値の範囲を回復するために，頻度テーブルの names 属性を数値に強制変換していること注意せよ．この問題を for ループ (Chapter 9 [ループと条件付き実行], page 40 で議論する) を使って解く「明白な」やり方はあまりに非能率的で実用的ではない．

また，そのような行列の約 20 に 1 つが特異であることはおそらく驚きであろう．

5.6 配列の一般化転置

関数 aperm(a, perm) は配列 a を置換するために使うことができる．引数 perm は整数 $\{1, \ldots, k\}$ のある置換でなければならない，ここで k は a の添字の数である．この関数の結果は a と同じサイズの配列であるが perm[j] で与えられるもとの次元が j 番めの新しい次元になっている．この操作は行列の転置の一般化と考えるのが最もわかりやすいであろう．実際，もし A が行列 (つまり，2 重の添字を持つ配列) ならば

```
> B <- aperm(A, c(2,1))
```

で与えられる B はまさに A の転置である．この特別な場合には似たような関数 t() を使うことが出来，B <- t(A) としても良い．

5.7 行列に対する諸機能

先に述べたように行列はまさに 2 重の添字を持つ配列のことである．しかし行列は重要な特殊例であるため別個の議論が必要となる．R には行列専用の多くの演算と関数がある．例えば先に述べたように t() は行列の転置関数である．関数 nrow(A) と ncol(A) は行列 A のそれぞれ行数と列数を与える．

5.7.1 行列の積

演算子 %*% は行列の積を求めるのに使われる．$n\times 1$ 行列もしくは $1\times n$ 行列は文脈からそうするのが適当ならば，勿論 n-ベクトルとして使える．逆に行列の積演算に現れるベクトルは，行列の積が意味を持つ限り，行ベクトルか列ベクトルに自動的に変換解釈される (後で見るように，これはいつでも曖昧さなしで可能なわけではない)．

例えば，A と B が同じサイズの正方行列とすると

```
> A * B
```

は成分毎の積からなる行列であり

```
> A %*% B
```

は行列積である．もし x がベクトルなら

```
> x %*% A %*% x
```

は 2 次形式を表す．[1]

関数 crossprod() は「クロス積」を作る，つまり crossprod(X, y) は t(X) %*% y と同じであるが，より効率的である．もし crossprod() の第 2 引数が省略されると，第 1 変数と同じと解釈される．

diag() の意味は引数によって異なる．diag(v) は対角成分が引数ベクトルであるような対角行列を表す，ここで v はベクトルである．diag(matrix) は引数行列の主対角成分からなるベクトルを表す．これは Matlab の diag() と同じ便法である．また少々混乱を招きかねないが，k が 1 つのスカラーならば diag(k) は k× k 単位行列を表す！

5.7.2 線形方程式と逆行列

線形方程式をを解くということは行列積の逆の操作である．

```
> b <- A %*% x
```

のあとで，A と b のみが与えられたとき，ベクトル x がこの線形連立方程式の解である．R では，

[1] 表現 x %*% x は曖昧であることを注意しよう．なぜならこれは $x'x$ もしくは xx' の双方を意味し得るから (x は列ベクトルとする)．こうした場合は暗黙のうちにより小さくなる行列が選ばれる．したがってこの場合にはスカラー $x'x$ が結果となる．行列 xx' は cbind(x) %*% x，もしくは x %*% rbind(x) として計算できる．なぜなら cbind() もしくは rbind() の結果は常に行列を表すからである．

```
> solve(A,b)
```

によってこれを解いて x を得ることができる (精度を多少失うが) . 線形代数の式で表すと $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ である . ここで $\mathbf{A}^{-1}$ は

$\mathbf{A}$ の「逆行列」であって ,

```
solve(A)
```

によって計算することもできるが , 必要になることはまれである . 数値計算上は , solve(A,b) の代わりに x <- solve(A) %*% b を使用することは効率が悪くまた不安定である .

多変量計算のときに使われる 2 次形式 $\mathbf{x}'\mathbf{A}^{-1}\mathbf{x}$ は , A の逆行列を計算するより , x %*% solve(A,x) などのように計算するべきである .

5.7.3 固有値と固有ベクトル

関数 eigen(Sm) は対称行列 Sm の固有値と固有ベクトルを計算する . この関数の結果は values と vectors と言う名前の成分を持つリストである . 付値

```
> ev <- eigen(Sm)
```

はこのリストを ev に代入する . ev$val は Sm の固有値で , ev$vec は対応する固有ベクトルの行列である . もし固有値だけが欲しいのなら次のような付値でも良い :

```
> evals <- eigen(Sm)$values
```

evals は固有値のベクトルを含み , 第 2 成分は捨てられる . もし表現

```
> eigen(Sm)
```

がそれ自身命令として使われるならば , 2 つの成分が名前付きで出力される .

5.7.4 特異値分解と行列式

関数 svd(M) は任意の行列引数 M を取り , M の特異値分解を計算する . これは M と同じ列空間を持ち列が直交する行列 U , その列空間が M の行空間と一致し列が直交する第 2 の行列 V , そして正の要素を持つ対角行列 D からなり , M = U %*% D %*% t(V) が成立する . D は実際は対角成分からなるベクトルとして返される . svd(M) の結果は実際には自明な名前 d, u そして v という 3 つの成分を持つリストである .

もし M が正方行列ならば ,

```
> absdetM <- prod(svd(M)$d)
```

が M の行列式の絶対値を計算することが容易に分かる . もしこの計算が様々な行列に対してしばしば必要なら , 次のような R の関数

```
> absdet <- function(M) prod(svd(M)$d)
```

を定義すれば , その後は absdet() をまさに他の R の関数であるかのように使える . 更に自明であるがひょっとすると役に立つかも知れない例として , 正方行列のトレースを計算する関数 , tr() としよう , を考えてみよ [ヒント : ループをまともに使う必要は無い . 関数 diag() を見よ .]

5.7.5 最小自乗法による当てはめと QR 分解

関数 lsfit() は最小自乗法による当てはめの結果を与えるリストを返す . 次のような付値

```
> ans <- lsfit(X, y)
```

は最小自乗法による当てはめの結果を与える．ここで y は観測値のベクトル，X は計画行列である．詳細に付いてはヘルプ機能を見よ，また同様に随伴する関数 ls.diag() を特に参照すること．定数項が自動的に含められること，したがって X の列として明示的に含める必要が無いことを注意せよ．

もう 1 つの関係の深い関数は qr() とその仲間である．次の付値を考えよう．

```
> Xplus <- qr(X)
> b <- qr.coef(Xplus, y)
> fit <- qr.fitted(Xplus, y)
> res <- qr.resid(Xplus, y)
```

これらは y の X の範囲の上への直交射影を fit に，直交補空間上への射影を res に，射影の係数ベクトルを b に計算して与える．つまり，b は本質的に Matlab の 'backslash' 演算子の結果である．

X がフルランクであることは仮定されていない．冗長性は発見され，見付かれば取り除かれる．

この代替物は最小自乗法を行うより古い低水準の方法である．ある場合には依然として有用であるが，Chapter 11 [R における統計モデル], page 50 で議論されるように，現在は統計モデル機能で一般的に置き換えられている．

5.8 区分化された行列を作る，cbind() と rbind()

すでに非公式に見てきたように，行列は他のベクトルや行列から関数 cbind(), rbind() を用いて作ることが出来る．簡単にいうと，rbind() は幾つかの行列を水平 (列順) に並べて，cbind() は垂直 (行順) に並べて行列を作り出す．

付値

```
> X <- cbind(arg_1, arg_2, arg_3, ...)
```

において cbind() の引数は，任意の長さのベクトルであるか，または同じ列サイズの行列である必要がある．結果は連結された引数 *arg_1*,*arg_2*, ... が列となる行列である．

もし cbind() の引数の一部がベクトルで，他のすべての行列引数の (同一の) 列サイズより短い時には，それらは行列の列サイズ (行列の引数が全く無い時は最長のベクトルの長さ) に一致するまでリサイクル使用される

関数 rbind() は行に対して同様の操作を行なう．この場合，すべてのベクトル引数は，必要なら循環的に延長されて，行ベクトルとして扱われる．

X1 と X2 が同じ数の行を持つとしよう．これらを列方向に結合して行列 X を作り，更に第 1 列に 1 だけからなる列を付け加えるには次のようにする．

```
> X <- cbind(1, X1, X2)
```

rbind() と cbind() の結果は常に行列の属性を持つ．したがってベクトル x を 1 列もしくは 1 行だけからなる行列と特に指定したければ，それぞれ cbind() や rbind() を使うのがおそらく最も単純な方法であろう．

5.9 配列に対する結合関数 c()

cbind() や rbind() が dim 属性を尊重する連結関数だとすれば，基本関数 c() はむしろ，数値からなるオブジェクトからすべての dim 属性と dimnames 属性を取り払ってしまうことを注意しよう．このことは，それはそれで，しばしば役に立つ性質である．

配列を単純なベクトルオブジェクトに強制変換する公式の方法は as.vector() を使うことである．

```
> vec <- as.vector(X)
```

しかしながら同様の結果は c() をたった 1 つの引数で用いることにより単なる副作用として達成できる：

```
> vec <- c(X)
```

両者の間には若干の差異があるが，結局のところどちら (前者が望ましいが) を使うかは趣味の問題である．

5.10 因子から度数分布表を作る

因子によりグループへの分割が定義されることを思いだそう．同様に対になった因子は 2 元分割を定義する，等々．関数 table() は等しい長さの因子から頻度表を計算することを可能にする．もし k 個のカテゴリが引数として与えられれば，結果は k-元の頻度配列である．

例えば statef がデータベクトルの各項目に対する状態コードを与える因子とする．付値

```
> statefr <- table(statef)
```

は statefr に標本中の各コードの頻度表をあたえる．頻度はカテゴリの水準により順序付けられるとともにラベルが付けられる．この単純な例は，より便利だが，次の表現と同値である．

```
> statefr <- tapply(statef, statef, length)
```

更に incomef がデータベクトル中の各項目に対して適当に定義 (例えば cut() 関数を用い) された「所得クラス」を与えるカテゴリとしよう：

```
> factor(cut(incomes, breaks = 35+10*(0:7))) -> incomef
```

すると頻度の 2 元分割表を計算するには次のようにする：

```
> table(incomef,statef)
         statef
incomef   act nsw nt qld sa tas vic wa
  (35,45]   1   1  0   1  0   0   1  0
  (45,55]   1   1  1   1  2   0   1  3
  (55,65]   0   3  1   3  2   2   2  1
  (65,75]   0   1  0   0  0   0   1  0
```

より多元の頻度表への拡張は容易である．

6 リストとデータフレーム

6.1 リスト

R の「リスト (*list*)」とはオブジェクトの順序付けられた集まりからなるオブジェクトのことである．個々の成分オブジェクトは，その「成分 (*component*)」と呼ばれる．

成分は同じモード・型である必要は特に無く，例えば 1 つのリストは数値ベクトル，論理ベクトル，行列，複素数ベクトル，文字配列，関数，その他から成り立っていても良い．次はリストを作る簡単な例である：

```
> Lst <- list(name="Fred", wife="Mary", no.children=3,
              child.ages=c(4,7,9))
```

成分は常に「順番が付けられ」，常にそれで参照出来る．したがって，もし `Lst` が 4 つの成分を持つリストならば，これらは個別に `Lst[[1]]`, `Lst[[2]]`, `Lst[[3]]`, `Lst[[4]]` と参照することが出来る．もし更に `Lst[[4]]` がベクトルなら `Lst[[4]][1]` はその最初の項目である．

もし `Lst` がリストならば，関数 `length(Lst)` はそれが含む (トップレベルの) 成分の個数を与える．

リストの成分には「名前 (*name*)」を付けることが出来，その場合はその成分は，2 重鍵括弧の中に順番の代わりに成分の名前を文字列で与えたり，より簡単には次の形の表現で参照することが可能である

> *name*$*component_name*

これは，もし番号を忘れても正しい成分を得ることの出来る，非常に便利な記法である．

したがって上に挙げた簡単な例では：

`Lst$name` は `Lst[[1]]` と同じであり，文字列 `"Fred"` となる，

`Lst$wife` は `Lst[[2]]` と同じであり，文字列 `"Mary"` となる,

`Lst$child.ages[1]` は `Lst[[4]][1]` と同じであり，数 4 になる．

更に，リスト成分の名前を 2 重鉤括弧の中に使うことができる，つまり， `Lst[["name"]]` は `Lst$name` と同じである．これは抜き出されるべき成分の名前が，別の変数に次のように入っているときに特に便利である．

```
> x <- "name"; Lst[[x]]
```

`Lst[[1]]` と `Lst[1]` を区別することは非常に重要である. ‘`[[...]]`’ は 1 つの要素を抜き出す演算子であり，他方で ‘`[...]`’ は一般的な添字指定演算子である．したがって，前者は「リスト `Lst` 中の最初のオブジェクト」であり，もし名前付のリストであってもその名前は含まれない．後者は「リスト `Lst` の最初の項目のみからなるサブリストであり，もしそれが名前付のリストならば，該当する名前がサブリストに伝わる」.

成分の名前は，それらを一意的に特定するのに必要な最小の数の文字列に省略することが出来る．したがって `Lst$coefficients` は `Lst$coe` で，`Lst$covariance` は `Lst$cov` で最も簡略に特定することが出来る．

名前のベクトルは実際には他と同様に単にリストの属性であり，そうしたものとして扱うことが出来る．リスト以外の構造も勿論同様に「名前」属性を与えることが出来る．

6.2 リストを作る，変える

新しいリストは，すでに存在するオブジェクトから関数 `list()` で生成することが出来る．次の形式の付値

```
> Lst <- list(name_1=object_1, ..., name_m=object_m)
```

は m 個のオブジェクト *object_1*, ..., *object_m* を成分に使ったリスト `Lst` を生成し，各オブジェクトの名前を名前ベクトル (自由に選べる) として持つ．もしこれらの名前が無ければ成分は単に順序付けられるだけである．リストを形成する成分は新しいリストを作る際に「コピー」され，オリジナルは変化しない．

リストは，他の添字を持つオブジェクトと同様に，追加の成分を指定して拡張できる．例えば，

```
> Lst[5] <- list(matrix=Mat)
```

6.2.1 リストを結合する

連結関数 `c()` がリストの引数を与えられると，結果はリストモードのオブジェクトになり，その成分は引数リストのそれらが順につなげられたものになる．

```
> list.ABC <- c(list.A, list.B, list.C)
```

ベクトルオブジェクトを引数にとると，連結関数は同様にすべての引数を，単一のベクトル構造につなぎ合わせたことを思いだそう．この場合 `dim` 属性等，他のすべての属性は捨てられる．

6.3 データフレーム

「データフレーム (*data frame*)」とは `data.frame` クラスを持つリストのことである．データフレームに入れることの出来るリストには幾つかの制限がある，つまり

- 成分はベクトル (数値，文字，もしくは論理値)，因子，数値行列，リスト，もしくは他のデータフレームに限られる，
- 行列，リスト，そしてデータフレームは，それらがそれぞれ持つ列，要素，変数と同じ数の変数を新しいデータフレームに付け加える，
-
  数値ベクトルと論理値，因子はそのままの状態で含まれ，文字ベクトルは因子に強制変換される．その水準はベクトルに現れるユニークな値である，
- データフレームに変数として現れるベクトル構造は，すべて同じ「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，

データフレームは多くの点で，異なったモードや属性を持ち得る列を持つ行列と見倣すことが出来る．それは行列形式で表示できるし，その行や列は行列の添字形式で抜き出すことが出来る．

6.3.1 データフレームを作る

データフレームの列 (成分) に対する要請を満たすオブジェクトは，関数 `data.frame` を用いてデータフレームの作成に用いることが出来る．

```
> accountants <- data.frame(home=statef, loot=income, shot=incomef)
```

リストの成分がデータフレームの要請を満たせば，関数 `as.data.frame()` を用いてデータフレームに強制変換出来る．

データフレームを新規に作る最も簡単な方法は read.table() 関数を用いて，外部のファイルからデータフレームの全体を読み込むことである．これに付いては更に Chapter 7 [ファイルからデータを読込む], page 30 で議論される．

6.3.2 関数 attach() と detach()

リストのコンポーネントに対する accountants$statef といった $-記法はいつでも便利というわけではない．有用な機能はリストやデータフレームの成分を，毎回リスト名を明示的に引用する必要なしに，成分名の下に変数として一時的に目に見えるようにすることであろう・

関数 attach() は，その引数と同じ名前を持つディレクトリを用意することにより，データフレームを作ることが出来る．例えば lentils が 3 つの変量 lentils$u, lentils$v, lentils$w を持つデータフレームとしよう．(検索リストへの) 追加

```
> attach(lentils)
```

はこのデータフレームを探索リストの第 2 番目の位置に置き，探索リストの第 1 番目の位置に変数 u, v, w が無いという条件の下で，u, v, w をデータフレーム lentils の 3 つの変量名として使うこと使うことができるようになる．ここに於いて次の付値

```
> u <- v+w
```

はデータフレームの成分 u を置き換えずに，むしろ検索リストの第 1 位置の作業領域にある別の変数 u からそれを隠蔽する．データフレーム自身に永続的な変更を加えたければ，最も単純な方法はもう一度$-記法に戻ることである：

```
> lentils$u <- v+w
```

しかしながら成分 u の新しい値は (検索リストから) 外し，もう一度追加するまで見えるようにはならない．

データフレームを (検索リストから) 外すには次の関数を用いる．

```
> detach()
```

より正確には，この宣言は現在 第 2 位置 にあるものを検索リストから外す．従って，現在の例では，lentils$u といったリスト記法を用いない限り，u, v, w はもはや見ることは出来なくなる．

検索パス上 2 より大きな位置にある存在は detach にその番号を引数として与えれば抹消出来るが，いつも名前を使うほうが安全である．例えば detach(lentils) とか detach("lentils") といった風にする．

> ノート：現在の R のリリースでは検索リストは最大で 20 の項目を含むことが出来る．同じデータフレームを重ねて追加しないようにしよう．変量の使用が終ったらすぐにデータフレームを外そう．
>
> ノート：現在の R のリリースでは，リストやデータフレームは位置 2 もしくはそれ以上にしか登録できない．登録されたリストやデータフレームに直接付値することはできない (従って，ある程度それらは静的である)．

6.3.3 データフレームを使った作業

同じ作業ディレクトリで多くの異なった問題を快適に処理すること出来るようにする有用な指針は

- 適切に定義され分離された問題に対するすべての変量を，1 つのデータフレームに集積し，それに適当な意味が明瞭な名前をつけよ；

- ある問題を処理する時は，適当なデータを検索リストの第 2 位置に追加し，第 1 位置を作業用や一時的な変数のために使え；
- 問題を終了する前に，将来の参考用に保存しておきたいすべての変量を $-記法を用いてデータフレームに追加し，それから detach() せよ；
- 最後にすべての不要な変量を作業ディレクトリから消し去り，忘れ残しの一時的変量がないように可能な限りきれいにしておけ．

このようにすれば例えば，同じディレクトリでどれもが x, y, z という変数名を持つ多くの問題を処理することは極めて簡単になる．

6.3.4 任意のリストの登録

attach() は単にディレクトリやデータフレームを検索リストに登録するだけで

なく，他のオブジェクトのクラスの登録も出来る一般的な関数である．特にモードが "list" の任意のオブジェクトは同じように登録できる：

```
> attach(any.old.list)
```

登録されたものは全て，位置番号や，より好ましくは名前を使って，detach で抹消できる．

6.3.5 検索パスを操作する

関数 search は現在の検索 パスを示し，従ってどのようなデータフレームやリスト (そしてパッケージ) が追加・除去されたかを追跡するのに非常に役に立つ．最初それは次のものを与える

```
> search()
[1] ".GlobalEnv"   "Autoloads"    "package:base"
```

ここで .GlobalEnv は作業スペース[1] である．

lentils が加えられた後では

```
> search()
[1] ".GlobalEnv"   "lentils"      "Autoloads"    "package:base"
> ls(2)
[1] "u" "v" "w"
```

のようになり，例にあるように ls (または objects) を使って検索 パス上の任意位置の中身を調べることができる．

最後に，データフレームを取り除き，それが検索パスから除かれていることを確認する．

```
> detach("lentils")
> search()
[1] ".GlobalEnv"   "Autoloads"    "package:base"
```

[1] 第 2 項の意味については autoload に対するオンラインヘルプを見よ．

7 ファイルからデータを読込む

大規模なデータオブジェクトは，普通 R のセッション中にキーボードから入力されるよりは，外部ファイルから値として読み込まれる．R の入力機能は単純で，その機能はかなり厳格で融通がきかない．R の設計者は，使用者が入力ファイルをエディタや

Perl を用いて， R の要求を満たすように入力ファイルを変形[1]

できるであろうことを当然の前提としている．普通これは非常に単純である，

もし変量が (我々が強く勧めるように) 主にデータフレームに収められるなら，それらはデータフレームの全体が `read.table()` 関数を用いて直接読み込めるようになっているべきである．又よりプリミティブな入力関数 `scan()` があり，直接呼び出すことが出来る．

データの R への取り込み，同じく取り出しについてより詳しいことはマニュアル *R Data Import/Export* を見よ．

7.1 `read.table()` 関数

データフレームの全体を直接読み込むためには，外部ファイルは普通特別な形を持つ．

- ファイルの第 1 行はデータフレーム中の各変量に対する「名前」を含むべきである，
- ファイルのその他の各行は最初の項目として各変量に対する「行ラベル」と値を持つべきである

もしファイルが最初の行に，第 2 行よりも 1 つ少ない項目を持つなら，こうした配置がされていると仮定される．データフレームとして読み込むべきファイルの最初の数行は，例えば次のようになる．

名前と行ラベルを持つ入力ファイルの書式：

```
     Price    Floor     Area   Rooms     Age  Cent.heat
01   52.00    111.0      830     5       6.2       no
02   54.75    128.0      710     5       7.5       no
03   57.50    101.0     1000     5       4.2       no
04   57.50    131.0      690     6       8.8       no
05   59.75     93.0      900     5       1.9      yes
...
```

既定として (行ラベルを除くと) 数値項目は数値変量として読み込まれ，上の例の `Cent.heat` のような非数値変量は因子として読み込まれる．必要ならこれは変更できる．

関数 `read.table()` はデータフレームを直接読み込むために使うことが出来る．

```
> HousePrice <- read.table("houses.data")
```

しばしば行ラベルを含めることをやめ，既定のラベルを使いたいかも知れない．こうした場合，ファイルは行ラベルの列を次のように省略出来る：

[1] Unix では Sed や Awk を用いることが出来る．

```
行ラベルを持たない入力ファイルの書式：

Price    Floor     Area   Rooms     Age  Cent.heat
52.00    111.0      830     5       6.2      no
54.75    128.0      710     5       7.5      no
57.50    101.0     1000     5       4.2      no
57.50    131.0      690     6       8.8      no
59.75     93.0      900     5       1.9     yes
...
```

そうするとデータフレームは次のように読み込むことが出来る：

```
> HousePrice <- read.table("houses.data", header=TRUE)
```

ここでオプション heading=T は最初の行は見出しの行であること，従って，ファイルの書式の含意から，明白な行ラベルは与えられていないことが指示されている．

7.2 scan() 関数

データベクトルが同じ長さで，並列的に読み込まれるべきであると仮定しよう．更に，3 つのベクトルがあり，最初は文字モードで残りの 2 つは数値モードとし，ファイルは input.dat とする．最初のステップは scan() を用い，この 3 つのベクトルをリストとして次のように読み込むことである：

```
> inp <- scan("input.dat", list("",0,0))
```

2 つ目の引数は，読み込むべき 3 つのベクトルのモードを確定するための，ダミーのリストである．inp に収められた結果は，読み込まれた 3 つのベクトルをコンポーネントに持つリストである．データ項目を 3 つの分離されたベクトルに分離するためには，次のような付値を使う：

```
> label <- inp[[1]]; x <- inp[[2]]; y <- inp[[3]]
```

より簡便に，ダミーリストは名前付のコンポーネントを持つことが出来，その時はその名前を読み込まれるベクトルにアクセスするために使うことが出来る：

```
> inp <- scan("input.dat", list(id="", x=0, y=0))
```

もし変量に個々にアクセスしたければ，それらをワーキングフレーム中で変量に再付値するか：

```
> label <- in$id; x <- in$x; y <- in$y
```

リストを探索リストの第 2 位置に置く (Section 6.3.4 [任意のリストの登録], page 29 を見よ)．

もし第 2 引数が単一の変数でリストでなければ，単独のベクトルが読み込まれ，そのすべてのコンポーネントはダミー変数と同じモードでなければならない．

```
> X <- matrix(scan("light.dat", 0), ncol=5, byrow=TRUE)
```

より洗練された読み込み機能があり，このマニュアルの中で詳しく述べられるであろう．

7.3 組み込みデータセットにアクセスする

50 を越えるデータセットが R とともに提供されており，その他のものがパッケージ (R とともに提供される標準パッケージを含み) から利用できる．S-Plus と異なり，これらのデータセットは関数 data を用い明示的にロードされなければならない．基本システム中のデータセットの一覧を見るには

```
data()
```

とし，それらの 1 つをロードするには，例えば次のようにする．

```
data(infert)
```

多くの場合これは同じ名前を持つ R のオブジェクト，普通データフレーム，をロードする．しかしながら，幾つかの場合それは複数のオブジェクトをロードする．何が起こるかはオブジェクトのオンラインヘルプを見よ．

7.3.1 他の R パッケージからデータを読み込む

他のパッケージのデータを利用するには，package 引数を利用せよ．例えば：

```
data(package="nls")
data(Puromycin, package="nls")
```

もしあるパッケージが library で付け加えられたら，そのデータセットは自動的に検索リストに含められ，従って

```
library(nls)
data()
data(Puromycin)
```

によって現在登録されている全てのパッケージ (少なくとも **base** と **nls**) 中の全てのデータセットが一覧でき，それから (もしそうしたデータセットが存在すれば) 最初のパッケージから Puromycin がロードされる．

ユーザが提供したパッケージはデータセットの豊富なソースになり得る．

7.4 データを編集する

データフレームか行列に対して呼び出されたときは，edit での編集は表計算形式になる．これは一旦データセットを読み込んだあとでちょっと変更するのに便利である．命令

```
> xnew <- edit(xold)
```

は自分のデータセット xold を編集することを可能にし，終了時には修正されたオブジェクトは xnew に付値される．

表計算形式のインターフェースによって新しいデータを入力するには

```
> xnew <- edit(data.frame())
```

のようにする．

8 確率分布

8.1 統計数値表としての R

R の 1 つの便利な利用法は，統計数値表の広範囲なセットを提供することである．累積分布関数 (CDF) $P(X \le x)$，確率密度関数，そしてクォンタイル関数 (q を与えたとき，$P(X \le x) > q$ となる最小の x)，更に分布に従う乱数をシミュレーションする関数が提供されている．

分布	R での名前	追加引数
ベータ (beta)	`beta`	`shape1, shape2, ncp`
2 項 (binomial)	`binom`	`size, prob`
コーシー (Cauchy)	`cauchy`	`location, scale`
カイ自乗 (chi-squared)	`chisq`	`df, ncp`
指数 (exponential)	`exp`	`rate`
F (F)	`f`	`df1, df1, ncp`
ガンマ (gamma)	`gamma`	`shape, scale`
幾何 (geometric)	`geom`	`prob`
超幾何 (hypergeometric)	`hyper`	`m, n, k`
対数正規 (log-normal)	`lnorm`	`meanlog, sdlog`
ロジスティック (logistic)	`logis`	`location, scale`
負の 2 項 (negative binomial)	`nbinom`	`size, prob`
正規 (normal)	`norm`	`mean, sd`
ポアッソン (Poisson)	`pois`	`lambda`
t (Student の t)	`t`	`df, ncp`
一様 (uniform)	`unif`	`min, max`
ワイブル (Weibull)	`weibull`	`shape, scale`
ウィルコクソン (Wilcoxon)	`wilcox`	`m, n`

ここで与えられた名前の頭に，‘d’ を付ければ密度関数，‘p’ を付ければ CDF, ‘q’ を付ければクォンタイル関数になる．そして ‘r’ を付ければ乱数をシミュレーションする．最初の引数は `d`*xxx* に対しては `x`，`p`*xxx* に対しては `q`，`q`*xxx* に対しては `p`，`r`*xxx* に対しては `n` (`rhyper` と `rwilcox` は例外で `nn`) である．非中心度パラメータ `ncp` は現在のところ CDF と幾つかの他の関数に対してだけ利用可能である：現況についての詳細はオンラインヘルプを見よ．

`p`*xxx* そして `q`*xxx* 関数はみな論理値引数 `lower.tail` と `log.p` を持ち，`d`*xxx* 関数は `log` を持つ．これは例えば，累積 (もしくは “integrated”)「危険 (*hazard*)」関数，$H(t) = -\log(1 - F(t))$ を

```
- pxxx(t, ..., lower.tail = FALSE, log.p = TRUE)
```

から求めたり，またはより正確な対数尤度を (`dxxx(..., log = TRUE)` により) 直接計算できるようにするためである．

更に正規分布からの標本のスチューデント化範囲の分布のための関数 `ptukey` と `qtukey` がある．

幾つかの例をあげる．

```
> ## t 分布に対する両側 p-値
> 2*pt(-2.43, df = 13)
> ## F(2, 7) 分布に対する上側 1% 点
> qf(0.99, 2, 7)
```

8.2 データのセットの分布を調べる

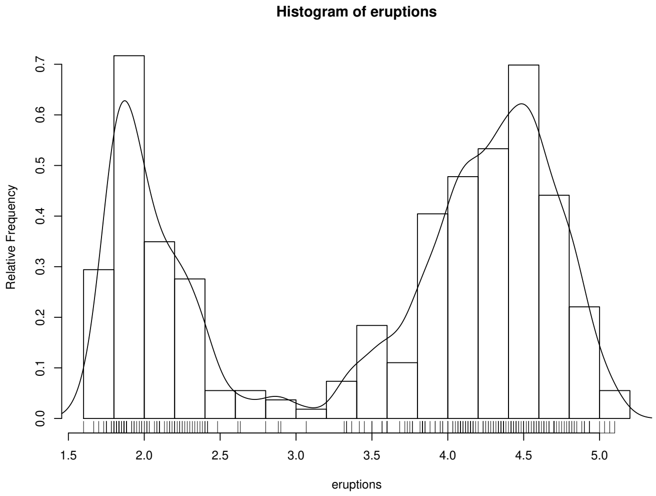
(1 変量の) データの集まりを与えると，その分布を多くの方法で調べることができる．最も簡単な方法はその度数を数えることである．2 つの少し異なる要約が `summary` と `fivenum` で与えられ，度数の表示は `stem` (“ 幹葉表示 (stem and leaf plot)”) で得られる．

```
> data(faithful)
> attach(faithful)
> summary(eruptions)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.600   2.163   4.000   3.488   4.454   5.100
> fivenum(eruptions)
[1] 1.6000 2.1585 4.0000 4.4585 5.1000
> stem(eruptions)

  The decimal point is 1 digit(s) to the left of the |

  16 | 070355555588
  18 | 000022233333335577777777888822335777888
  20 | 00002223378800035778
  22 | 0002335578023578
  24 | 00228
  26 | 23
  28 | 080
  30 | 7
  32 | 2337
  34 | 250077
  36 | 0000823577
  38 | 2333335582225577
  40 | 0000003357788888002233555577778
  42 | 03335555778800233333555577778
  44 | 02222335557780000000023333357778888
  46 | 0000233357700000023578
  48 | 00000022335800333
  50 | 0370
```

幹葉表示はヒストグラムに似ており，R はヒストグラムを表示する関数 `hist` を持つ．

```
> hist(eruptions)
# 区間幅をより小さくし，密度をプロットする
> hist(eruptions, seq(1.6, 5.2, 0.2), prob=TRUE)
> lines(density(eruptions, bw=0.1))
> rug(eruptions) # 実際のデータ点を示す
```

より洗練された密度プロットが density で得られ，density が作った線をこの 例に加える．バンド幅 bw は既定値では滑らかすぎるので，試行錯誤で得た (“興味ある” 密度では普通そうなる)．(バンド幅の自動選択法がパッケージ **MASS** と **KernSmooth** にある．)

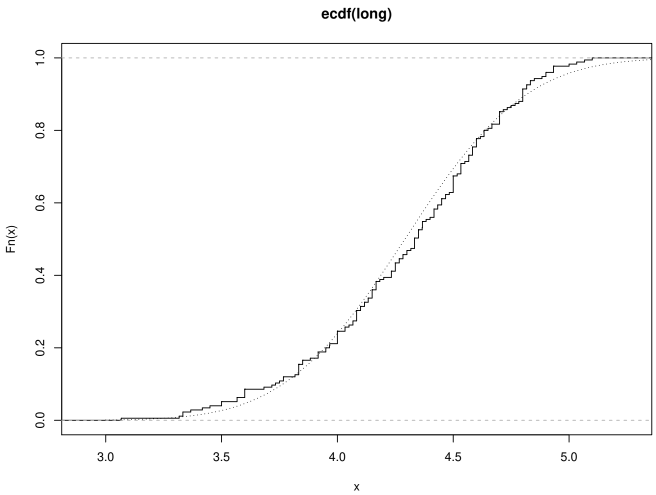


経験累積分布関数を標準パッケージ **stepfun** 中の関数 ecdf を使って書くことができる．

```
> library(stepfun)
> plot(ecdf(eruptions), do.points=FALSE, verticals=TRUE)
```

この分布は明らかに全ての標準的分布からはるかにずれている．右側，例えば 3 分以上の爆発，はどうだろうか? 正規分布を当てはめ，当てはめられた CDF を上書きしてみる．

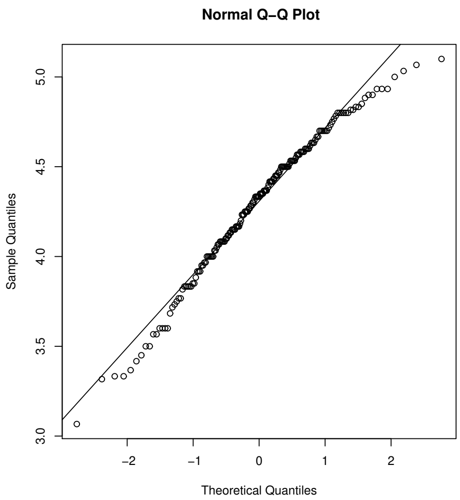
```
> long <- eruptions[eruptions > 3]
> plot(ecdf(long), do.points=FALSE, verticals=TRUE)
> x <- seq(3, 5.4, 0.01)
> lines(x, pnorm(x, mean=mean(long), sd=sqrt(var(long))), lty=3)
```



Q-Q (quantile-quantile) プロットでもう少し注意深く調べてみることができる．

```
par(pty="s")
qqnorm(long); qqline(long)
```

これはかなり良い当てはめを示すが，正規分布から期待されるよりは右側の裾が短い．これを t 分布からのシミュレーションデータと比較してみよう．




```
x <- rt(250, df = 5)
qqnorm(x); qqline(x)
```

これ (ランダム標本である) は普通正規分布よりは長い裾を持つ．生成された分布に対し Q-Q プロットを書いてみる．

```
qqplot(qt(ppoints(250), df=5), x, xlab="Q-Q plot for t dsn")
qqline(x)
```

最後に，正規性の公式的テストをしてみたくなる (またはしてみたくない) かも知れない．パッケージ **ctest** は Shapiro-Wilk 検定を提供する．

```
> library(ctest)
> shapiro.test(long)

         Shapiro-Wilk normality test

data:  long
W = 0.9793, p-value = 0.01052
```

そして Kolmogorov-Smirnov 検定も使える．

```
> 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
```

(我々は同じ標本から正規分布のパラメータを推定したので，分布論はここでは正確ではない．)

8.3 1 標本・2 標本検定

これまでは単一の標本を正規分布と比較して来た．より普通の操作は 2 つの標本の特徴を比較することである．

R では，下で使われているものを含む「古典的な」検定はすべて **ctest** パッケージの中にあることに注意．従って，これらを利用するにはパッケージを `library(ctest)` によって明示的に読む込む必要があるかもしれない．

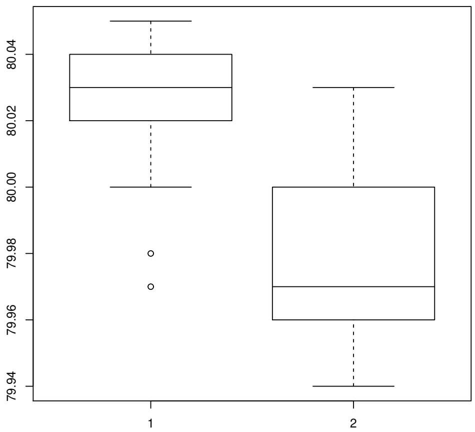
次の氷の溶解時の潜熱 (*cal/gm*) に関するデータの組を考えよう (Rice, 1995, p.490)．

```
Method A: 79.98 80.04 80.02 80.04 80.03 80.03 80.04 79.97
          80.05 80.03 80.02 80.00 80.02
Method B: 80.02 79.94 79.98 79.97 79.97 80.03 79.95 79.97
```

箱型図は 2 つの標本の簡単な図を用いた比較を提供する．

```
A <- scan()
79.98 80.04 80.02 80.04 80.03 80.03 80.04 79.97
80.05 80.03 80.02 80.00 80.02

B <- scan()
80.02 79.94 79.98 79.97 79.97 80.03 79.95 79.97

boxplot(A, B)
```

これは最初のグループが第 2 のグループより高い結果を与える傾向を示している．



2 つの標本の平均の同等性をテストするために，次のように (*unpaired*) t-検定を使うことができる．

```
> library(ctest)
```

```
> t.test(A, B)

         Welch Two Sample t-test

data:  A and B
t = 3.2499, df = 12.027, p-value = 0.00694
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.01385526 0.07018320
sample estimates:
mean of x mean of y
 80.02077  79.97875
```

これは正規性を仮定した上で，有意な差を示している．既定では R の関数は (S-PLUS の `t.test` とは異なり)2 標本の分散の同等性を仮定しない．

もし 2 つの標本が正規母集団からのものであれば，F 検定によって分散の同等性をテストできる．

```
> var.test(A, B)

         F test to compare two variances

data:  A and B
F = 0.5837, num df = 12, denom df =  7, p-value = 0.3938
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.1251097 2.1052687
sample estimates:
ratio of variances
         0.5837405
```

これは有意な差の証拠を示していない．したがって分散の同等性を仮定した古典的な t-検定を使うことができる．

```
> t.test(A, B, var.equal=TRUE)

         Two Sample t-test

data:  A and B
t = 3.4722, df = 19, p-value = 0.002551
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.01669058 0.06734788
sample estimates:
mean of x mean of y
 80.02077  79.97875
```

これら全ての検定は 2 標本の正規性を仮定している．2 標本 Wilcoxon (または Mann-Whitney) 検定は帰無仮説として同一の連続分布だけを仮定している．

```
> wilcox.test(A, B)

         Wilcoxon rank sum test with continuity correction

data:  A and B
W = 89, p-value = 0.007497
alternative hypothesis: true mu is not equal to 0

Warning message:
Cannot compute exact p-value with ties in: wilcox.test(A, B)
```

警告に注意しよう：各標本には幾つかの同一値があり，これはデータが離散分布からのもの (おそらく丸めによる) であることを強く示唆している．

2 標本をグラフで比較する幾つかの方法がある．既に箱型図の対を見て来た．次の

```
> library(stepfun)
> plot(ecdf(A), do.points=FALSE, verticals=TRUE, xlim=range(A, B))
> plot(ecdf(B), do.points=FALSE, verticals=TRUE, add=TRUE)
```

は 2 つの経験 CDF を示し，`qqplot` は 2 標本の Q-Q プロットを示す．コルモゴロフ-スミルノフ検定 (Kolmogorov-Smirnov test) は同一の連続分布を仮定して，2 つの経験 CDF の最大の食い違いを見る．

```
> ks.test(A, B)

         Two-sample Kolmogorov-Smirnov test

data:  A and B
D = 0.5962, p-value = 0.05919
alternative hypothesis: two.sided

Warning message:
cannot compute correct p-values with ties in: ks.test(A, B)
```

9 グループ化，ループと条件付き実行

9.1 グループ化された表現式

R は，その唯一の命令の型が関数か結果を返す表現式であると言う意味で，表現式言語である．付値ですら，その結果が付値された値であると言う意味で表現式である．そしてそれは表現式が使える場所では，いつでも使うことができる．特に多重の付値が可能である．

命令は括弧に入れて一緒にグループ化できる, `{expr_1; ...; expr_m}`．この場合グループの返り値はグループ中の最後の表現式の返り値である．そうしたグループは同様に 1 つの表現式であるから，例えば，それ自身括弧に含めることができ，更に大きな表現式の一部分として使われる，等々．

9.2 制御文

9.2.1 条件付き実行：`if` 文

次の形式の条件付き実行が構成できる

```
> if (expr_1) expr_2 else expr_3
```

ここで *expr_1* は論理値を返さねばならず，そうすると表現式全体の結果は明白である．

“短縮形” 演算子 `&&` と `||` は `if` 文中の条件としてしばしば使われる．`&` と `|` はベクトルの要素毎に適用されるのに対し，`&&` と `||` は長さ 1 のベクトルに適用され，必要なときだけ第 2 引数を評価する．

構文 if/else のベクトル化版である `ifelse` 関数がある．これは `ifelse(condition, a, b)` の形を持ち，その最長の引数の長さを持つベクトルを返す．その要素は，もし `condition[i]` が真ならば `a[i]` で，さもなければ `b[i]` である．

9.2.2 繰り返し実行：`for` ループ，`repeat` と `while`

また `for` ループ構文があり，次の形を持つ

```
> for (name in expr_1) expr_2
```

ここで e `name` はループ変数で，*expr_1* はベクトル値表現式 (しばしば `1:20` のような数列)，そして *expr_2* はしばしばグループ化された表現式で，ダミー変数 *name* を用いて書かれた副表現式を持つ．*expr_2* は *name* が *expr_1* の返り値であるベクトル中の値を動くとき繰り返し評価される．

例として，`ind` がクラスを指示するベクトルと仮定し，各クラスの中で `x` に対する `y` のプロットを別々に表示しよう．ここでの 1 つの可能性は後で述べられる `coplot()` を使うことであり，因子の各水準に対応するプロットの配列を作り出す．これを行うもう 1 つの次のような方法は，全てのプロットを 1 つの画面に表示する：

```
> xc <- split(x, ind)
> yc <- split(y, ind)
> for (i in 1:length(yc)) {
    plot(xc[[i]], yc[[i]]);
    abline(lsfit(xc[[i]], yc[[i]]))
```

```
    }
```

(関数 `split()` は，カテゴリにより指示されるクラスに従い，より大きなベクトルを分割し，ベクトルのリストを作り出すことを注意しよう．これは有用な関数であり，箱型図で主に使われる．詳細に付いては `help` を参照のこと．)

> 警告：`for()` ループは R ではコンパイル型言語におけるよりも使われることが少ない．R では「オブジェクト全体」を考慮したコードの方が，より明解で早くなることの方が普通である．

その他のループ機能としては次のようなものがある

```
> repeat expr
```

文と

```
> while (condition) expr
```

文．

`break` 文は任意のループを，もしかすると異常に，終了させるのに使うことができる．これは `repeat` ループ文を終了させる唯一の方法である．

`next` 文は 1 つの特定のサイクルを中断し「次に」スキップするのに使うことができる．

制御文は Chapter 10 [自分自身の関数を書く], page 42 で議論される「関数 (*functions*)」との関連で使われることが最も多く，そこではもっと多くの例が登場する．

10 自分自身の関数を書く

これまでに非公式に見てきたように，R 言語はユーザが「関数 (*function*)」モードのオブジェクトを作ることを許す．これらは特別な内部形式で保管され，他の表現で使うことが出来る R の真の関数である．こうすることにより R 言語は力強さ，便利さ，優美さを著しく増し，有用な関数を書くことを学ぶことは，R の使用を快適かつ生産的にする主要な方法の 1 つである．

R システムの一部として提供される mean(), var(), postscript() といった関数のほとんどはそれ自身 R で書かれており，従ってユーザが定義した関数と実態としては差がないことを強調しておこう．

1 つの関数は次のような付値で定義される

```
> name <- function(arg_1, arg_2, ...) expression
```

ここで expression は，引数 arg_i を用いある値を計算する R の (普通グループ化された) 表現式である．この表現式の値は関数の返り値になる．

関数の呼び出しは普通 *name*(*expr_1*, *expr_2*, ...) の形を取り，関数呼び出しが許されるあらゆる所に登場することが出来る．

10.1 簡単な例

最初の例として 2 標本 t-統計量を計算する (細部がすべて見える) 関数を考えよう．勿論，同じことをもっと簡単に行なう別の方法もあるから，これは人工的な例である．

関数は次のように定義される：

```
> twosam <- function(y1, y2) {
    n1  <- length(y1); n2  <- length(y2)
    yb1 <- mean(y1);   yb2 <- mean(y2)
    s1  <- var(y1);    s2  <- var(y2)
    s <- ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)
    tst <- (yb1 - yb2)/sqrt(s2*(1/n1 + 1/n2))
    tst
  }
```

もしこの関数を定義すれば，2 標本 t-検定は次のような呼び出しで実行できるであろう

```
> tstat <- twosam(data$male, data$female); tstat
```

2 番目の例として Matlab の backslash 命令を直接模倣する関数を考えてみよう．これは，行列 X の列ベクトルの張る空間への，ベクトル y の直交射影の係数を返すものである．(これは回帰係数の最小自乗推定と普通呼ばれる．) これは通常 qr() 関数を用いて行なわれるのだろうが，これは直接使うのはしばしば少々技巧的に過ぎ，安全に使うために次のような簡単な関数を用意するのが割にあうであろう．

y を $n\times 1$ ベクトル，X を $n\times p$ 行列とすると，

$X\backslash y$ の定義は $(X'X)^{-}X'y$ になる．ここで，$(X'X)^{-}$ は $X'X$ の一般化逆行列である．

```
> bslash <- function(X, y) {
  X <- qr(X)
  qr.coef(X, y)
}
```

このオブジェクトが作られれば，それはすべてのオブジェクトと同様に永続的であり，次のような文等で使うことが出来る

```
> regcoeff <- bslash(Xmat, yvar)
```

等々．

R の古典的な関数 lsfit() はこの作業及びそれ以上のこと[1] を，巧妙に行なう．それは逆に関数 qr() と qr.coef() を，上述した部分の計算を行なうために，上のような少々直観に反する仕方で使っている．従ってこうしたことが頻繁に使われるようならば，この部分を使いやすい単純な形に独立させることには，恐らく多少の価値があるであろう．もしそうならば，いっそう使いやすくするために，行列に対する 2 項演算にしたくなるかも知れない．

10.2 新しい 2 項演算子を定義する

もし bslash() 関数が別の名前，つまり次のような形

%*anything*%

を持てば，それを表現式の中で関数形で無く，「2 項演算子 (*binary operator*)」として使うことも出来る．例えば，もし真中の文字として ! を使ったとする．関数の定義は

```
> "%!%" <- function(X, y) { ... }
```

のように始まるであろう (引用マークに使用に注意)．そうするとこの関数は X %!% y の様に使うことが出来る．(バックスラッシュ記号そのものの使用は，この文脈では特別な問題を起こすので，都合の良い選択ではない．)

行列の積演算子 %*% や，行列の外積演算子 %o% は，このようにして定義された 2 項演算子の別の例である．

10.3 名前付きの引数と既定値

Section 2.3 [規則的な数列の生成], page 8 で最初に注意したように，もし呼び出された関数への引数が “name=object” の形で与えられるなら，それらの順序は任意である．更に引数列は名前が無く，位置順序に依存する形で始まることが出来るが，その際は名前付引数は位置依存の引数列の後に置く．

従って，もし関数 fun1 が

```
> fun1 <- function(data, data.frame, graph, limit) {
    [function body omitted]
  }
```

と定義されたとする．

そうすると，関数は色々な方法で呼び出すことが出来る．例えば

```
> ans <- fun1(d, df, TRUE, 20)
> ans <- fun1(d, df, graph=TRUE, limit=20)
> ans <- fun1(data=d, limit=20, graph=TRUE, data.frame=df)
```

は皆同じことである．

多くの場合，引数には常識的に適当な既定値を与えることが出来る．この時，既定値でよければそれらは関数の呼び出しからすべて省略することが出来る．例えば，もし関数 fun1 が

```
> fun1 <- function(data, data.frame, graph=TRUE, limit=20) { ... }
```

と定義されたとすると，それは上の 3 つの例と同値な

[1] モデルの章で説明される方法を参照せよ．

```
> ans <- fun1(d, df)
```

という形で呼び出すことも出来るし，既定値の 1 つを変更する

```
> ans <- fun1(d, df, limit=10)
```

という風にも使うことが出来る．

重要な注意として，既定値は任意の表現式で良く，同じ関数への他の引数を含むことすら可能であり，ここでの我々の簡単な例のように，定数に制限されることもないことを述べておく．

10.4 引数 '...'

もう 1 つの頻繁に必要になる要請としては，関数の引数の設定を，他の関数に引き継がせることがある．例えば，多くの描画関数は関数 `par()` を利用し，`plot()` の様な関数はユーザがグラフィックス出力を制御するために，グラフィックスパラメータを `par()` に渡すことを可能にする．(`par()` 関数についての詳細は Section 12.4.1 [par() 関数], page 68 を見よ)．これは関数に追加引数 (文字通りに

'...' と表される) を含め，この引数を継承させることにより可能になる．以下の図に概略的な例をあげる．

```
fun1 <- function(data, data.frame, graph=TRUE, limit=20, ...) {
  [omitted statements]
  if (graph)
    par(pch="*", ...)
  [これ以下は省略]
}
```

10.5 関数中からの付値

「関数内で行なわれる全ての付値は局所的かつ一時的であり，関数から抜け出す時に失われる」ことを注意しよう．従って，付値 `X <- qr(X)` は呼び出しプログラム中の引数の値に影響しない．

R における付値のスコープを支配する完全な規則を理解するためには，評価の「フレーム (*frame*)」に精通する必要がある．これは決して難解では無いが，少々高等な話題であり，このノートではこれ以上触れることはしない．

もし関数中で，大局で永続的な付値を行ないたいならば 'superassignment' 演算子`<<-` を使うか，関数 `assign()` を使うことが出来る．詳細については `help` 文書を見よ．S-Plus のユーザは R では `<<-` が異なった意味を持つことを注意すべきである．これらは Section 10.7 [スコープ], page 46 でさらに議論される．

10.6 より進んだ例

10.6.1 ブロックデザインに於ける効率因子

平凡かも知れないがより完全な関数の例として，ブロック実験計画の効率因子を見い出すことを考えよう．(この問題については既に Section 5.3 [添字の配列], page 19 で触れた．)

ブロックデザインは 2 つの因子，例えば `blocks`(`b` 水準) と `varieties` (`v` 水準) からなる．もし R と K がそれぞれ $v\times\ v$ そして $b\times\ b$ の「繰り返し (*replications*)」 そして 「ブロックサイ

ズ (*block size*)」行列で，N が $b\times v$ の出現行列 (incidence matrix) ならば，効率因子 (effciency factor) は次の行列の固有値として定義される

$$E = I_v - R^{-1/2}N'K^{-1}NR^{-1/2} = I_v - A'A,$$

ここで $A = K^{-1/2}NR^{-1/2}$．この関数を書く 1 つの方法が以下に与えられている．

```
> bdeff <- function(blocks, varieties) {
    blocks <- as.factor(blocks)             # 少し安全にするため
    b <- length(levels(blocks))
    varieties <- as.factor(varieties)       # 少し安全にするため
    v <- length(levels(varieties))
    K <- as.vector(table(blocks))           # dim 属性を除く
    R <- as.vector(table(varieties))        # dim 属性を除く
    N <- table(blocks, varieties)
    A <- 1/sqrt(K) * N * rep(1/sqrt(R), rep(b, v))
    sv <- svd(A)
    list(eff=1 - sv$d^2, blockcv=sv$u, varietycv=sv$v)
  }
```

この場合，固有値ルーチンよりも特異値分解を使う方が数値的に少し好ましい．

この関数の結果は，最初の成分である効率因子だけでなく，ブロックと variety に対する canonical contrast も与えるリストである．なぜなら，これらは時おり付随的で有益な定量的情報だからである．

10.6.2 プリントされた配列から全ての名前を取り除く

大きな行列や配列を表示するためには，配列の名前や数を取り去った閉じた形で表示することがしばしば有用である．`dimnames` 属性を取り除くことでこの効果を得ることは出来ず，むしろ配列に空の文字列からなる `dimnames` 属性を与える必要がある．例えば行列 `X` を表示するには

```
> temp <- X
> dimnames(temp) <- list(rep("", nrow(X)), rep("", ncol(X))
> temp; rm(temp)
```

とする．これをいっそう簡便に行なうには，同じ結果を達成する「一括りにした」ものとしての，以下に示された関数 `no.dimnames()` を使う．これはまた，効率的で有用なユーザ定義関数が如何に簡潔であり得るかということを説明している．

```
no.dimnames <- function(a) {
  ## 簡潔な出力のために配列から全ての次元名を取り去る．
  d <- list()
  l <- 0
  for(i in dim(a)) {
    d[[l <- l + 1]] <- rep("", i)
  }
  dimnames(a) <- d
  a
}
```

この関数を定義すれば，配列を閉じた形で次のように表示出来る

```
> no.dimnames(X)
```

これは，値よりもパターンこそが興味の対象である大きな整数値配列に対し特に有益である．

10.6.3 再帰的な数値積分

関数は再帰的であることが出来，関数自身の中で関数を定義することが出来る．しかしながら，そうした関数，実際には変数，はもしそれらが検索リスト上にあった時にそうなるようには，より高い順位の評価フレームにある呼び出し関数により継承されないことを注意しよう．

以下にあげた例は 1 次元の数値積分を行なう素朴な方法である．被積分関数は積分範囲の両端と中央で評価される．もし台形を 1 つだけ使った台形公式の値が，台形を 2 つ使った台形公式の値と十分近ければ．後者が積分の値として返される．さもなければ同じプロセスが再帰的に同じパネルに対し適用される．結果は関数の評価が 1 次関数から遠い場所に集中する適応的な積分のプロセスである．しかしながら，重いオーバーヘッドが存在し，被積分関数が十分滑らかであるとともに評価が極めて難しい時にのみ，他のアルゴリズムと肩を並べることが出来る．

この例は同時に R プログラミングにおける小さなパズルを与える．

```
area <- function(f, a, b, eps = 1.0e-06, lim = 10) {
  fun1 <- function(f, a, b, fa, fb, a0, eps, lim, fun) {
    ## 関数 ‘fun1’ は ‘area’ の中からだけ見える
    d <- (a + b)/2
    h <- (b - a)/4
    fd <- f(d)
    a1 <- h * (fa + fd)
    a2 <- h * (fd + fb)
    if(abs(a0 - a1 - a2) < eps || lim == 0)
      return(a1 + a2)
    else {
      return(fun(f, a, d, fa, fd, a1, eps, lim - 1, fun) +
             fun(f, d, b, fd, fb, a2, eps, lim - 1, fun))
    }
  }
  fa <- f(a)
  fb <- f(b)
  a0 <- ((fa + fb) * (b - a))/2
  fun1(f, a, b, fa, fb, a0, eps, lim, fun1)
}
```

10.7 スコープ

この節における議論はこの文書の他の部分よりも少々より技術的である．しかしながら，それは S-Plus と R の主要な違いの 1 つを明らかにする．

関数の本体に現れる記号は 3 つのクラスに分けられる．形式的パラメータと局所変数と自由変数である．関数の形式的パラメータは関数の引数リストに現れるものである．これらの値は実際の関数引数を形式的パラメータに「結び付ける ($binding$)」プロセスにより決定される．局所変数は関数の本体中で表現の評価により決定される．形式的パラメータでも局所変数でも無いものは，自由変数と呼ばれる．自由変数はもしそれが付値されれば局所変数になる．次の関数定義を考えよう．

```
f <- function(x) {
  y <- 2*x
  print(x)
```

```
    print(y)
    print(z)
  }
```

この例で x は形式的パラメータ，y は局所変数，そして z は自由変数である．

R では自由変数の中身は，関数が作られた環境を最初に探すことにより決定される．これは「構文解析的スコープ (*lexical scope*)」と呼ばれる．先ず cube という名前の関数を定義しよう．

```
  cube <- function(n) {
    sq <- function() n*n
    n*sq()
  }
```

関数 sq 中の変数 n はその関数への引数ではない．従ってスコープ規則を用いて，それに結びつけられるべき値を決めなければならない．静的なスコープ (static scope) の下では，値は n という名前の大局的変数に結び付いた値である．構文解析式のスコープの下では，それは関数 cube にあたえられたパラメータである．なぜなら，それが関数 sq が定義された時に変数 n に実際に結びつけられたものだからである．S-Plus と R における評価の違いは，S-Plus が n という名前の大局的変数を探すのに，R は関数 cube が起動された時作られる環境中で n という名前の変数を先ず探すことにある．

```
  ## S における最初の評価
  S> cube(2)
  Error in sq(): Object "n" not found
  Dumped
  S> n <- 3
  S> cube(2)
  [1] 18
  ## R で評価された同じ関数
  R> cube(2)
  [1] 8
```

構文解析的スコープはまた関数に「様々に変わり得る状態 (*mutable state*)」を与えるのに使うことが出来る．次の例では R を用いていかに銀行勘定を物真似することが出来るかを示す．銀行勘定機能では，収支または合計を持ち，引き出し機能，預け入れ機能，そして現在の収支バランスを示す機能が必要になる．これを acount の中で 3 つの関数を生成し，それからそれらを含むリストを返すことにより達成する．account が呼び出されると，それは数値引数 total を取り，3 つの関数を含むリストを返す．これらの関数は total を含む環境で定義されるので，その値を利用することが出来る．

特殊な付値演算子 <<- を用いて total に付随する値を変えることが出来る．この演算子はシンボル total を含む環境を包括する環境を振り返り，もしそうした環境があればその環境の中で total の値を右辺値で置き換える．もしシンボル total を見つけることなく，大局的もしくは最上位の環境に到達すれば，それに対する変数が作られ，それに右辺値が付値[2] される．<<- が別の関数の返り値である関数の中で使われた時にのみ，ここで説明された特別な行動が起こる．

```
  open.account <- function(total) {
    list(
      deposit = function(amount) {
        if(amount <= 0)
```

[2] ある意味でこれは S-Plus における行動を真似ている．なぜなら S-Plus ではこの演算子は常に大局的変数を作り出すかそれに付値するからである．

```
        stop("Deposits must be positive!\n")
      total <<- total + amount
      cat(amount, "deposited.  Your balance is", total, "\n\n")
    },
    withdraw = function(amount) {
      if(amount > total)
        stop("You don't have that much money!\n")
      total <<- total - amount
      cat(amount, "withdrawn.  Your balance is", total, "\n\n")
    },
    balance = function() {
      cat("Your balance is", total, "\n\n")
    }
  )
}

ross <- open.account(100)
robert <- open.account(200)

ross$withdraw(30)
ross$balance()
robert$balance()

ross$deposit(50)
ross$balance()
ross$withdraw(500)
```

10.8 環境を自分好みにする

ユーザは自分の環境を幾つかの異なった方法でカスタマイズ出来る．サイトの初期化ファイルがあり，全てのディレクトリはそれ自身の初期化ファイルを持つことが出来る．特殊な関数 .First と .Last を利用することも出来る．

サイトの初期化ファイルの場所は環境変数 R_PROFILE の値が用いられる．もしこの変数が設定されていなければ，R のホームのサブディレクトリ etc 中にある Rprofile.site が使われる．

このファイルは R があなたのシステムで起動されるたびに実行したい

命令を含むべきである．2 つ目の個人的プロファイルのファイルである.Rprofile.site[3] は任意のディレクトリに置くことが出来る．もし R がそのディレクトリで起動されると，このファイルが初期設定用に読みとられる．このファイルは個々のユーザに，彼らの作業空間の制御仕様を与え，異なった作業ディレクトリでは異なった始動手順を持つことを許す．もし '.Rprofile' が開始ディレクトリに無ければ，R はユーザのホームディレクトリ中で '.Rprofile' ファイルを探し (もし存在すれば) それを使う．

上の 2 種類のプロファイル用ファイル中か .RData イメージ中の .First() という名前の任意の関数は特殊な挙動を持つ．それは R セッションの冒頭に自動的に実行され環境を初期化するのに用

[3] 従ってこのファイルは Unix の隠しファイルである．

いることが出来る．例えば以下の例中の定義はプロンプトを $ に変え，他の様々なことがらをセットアップして，それ以降のセッション中で所与とする．

このようにファイルの実行順序は Rprofile，.Rprofile そして .First() となる．後のファイル中の定義はそれ以前のファイル中の定義をマスクする．

```
> .First <- function() {
  options(prompt="$ ", continue="+\t")  # $ がプロンプト
  options(digits=5, length=999)         # custom numbers and printout
  x11()                                 # グラフィックス用
  par(pch = "+")                        # プロット文字
  source(file.path(Sys.getenv("HOME"), "R", "mystuff.R"))
                                        # 個人的パッケージ
  library(stepfun)                      # step function パッケージを使う
}
```

同様に .Last() 関数が定義されていれば，セッションのまさに最後に実行される．1 つの例が以下に与えられている．

```
> .Last <- function() {
  graphics.off()                        # 安全のために
  cat(paste(date(),"\nAdios\n"))        # 食事の時間?
}
```

10.9 クラス，総称的関数とオブジェクト指向

あるオブジェクトのクラスは「総称的 (*generic*) な」関数として知られているものにより，それがどう処理されるかを決める．反対に総称的な関数は「引数自身のクラスに固有の」ある作業や行動を引数に対して行なう．もし引数が如何なるクラス属性も持たないか，問題になっている総称的な関数に対し特別にあつらえているのではないクラスを持つならば，常に「既定の動作 (*default action*)」が用意されている．

例をあげるとわかりやすいであろう．クラスのメカニズムはユーザが特別な目的のために，総称的な関数をデザインし書く機能を提供する．総称的な関数の例としては，オブジェクトをグラフィック表示する plot()，各種の解析の要約のための summary()，統計モデルの比較のための anova() がある．

あるクラスを特殊な方法で処理することが出来る総称的関数の数は極めて多い．例えば，クラス data.frame のオブジェクトにある流儀で役立つことの出来る関数には

```
[      [[<-    any     as.matrix
[<-    model   plot    summary
```

がある．現在の完全なリストは methods() 関数を使って得ることが出来る．

```
> methods(class="data.frame")
```

逆に 1 つの総称的な関数が処理できるクラスの数も極めて多くなることがある．

例えば，plot() 関数は既定のメソッドと，クラス "data.frame" や "density"，factor"，その他のクラスのオブジェクトのための変種を持つ．完全なリストは再び methods() 関数を用いて得ることができる．

```
> methods(plot)
```

この機構の完全な議論は公式の参考文献を参照されたい．

11 R における統計モデル

この章では読者が統計学の方法論，特に回帰分析と分散分析にある程度なじみがあることを前提にしている．もっと後では，一般化線形モデルと非線形回帰について多少知識があるという，より注文の多い仮定をする．

広範囲な問題に応用される一般的な道具を作ることを可能にするために，統計モデルの当てはめるのための道具は十分に定義されている．R は統計モデルを極めて簡単に当てはめるために，一連の連係された機能を提供している．紹介で述べたように，出力は最小限であり，詳細を知るには情報抽出関数を呼び出さなければならない．

11.1 統計公式を定義する；公式

統計モデルに対する雛型 (template) は，独立で同型の誤差を持つ線形回帰モデル

$$y_i = \sum_{j=0}^{p} \beta_j x_{ij} + e_i, \qquad e_i \sim \mathrm{NID}(0, \sigma^2), \qquad i = 1, \ldots, n$$

である．行列の言葉ではこれは

$$y = X\beta + e$$

と書くことが出来る．ここで y は「応答ベクトル (*response vector*)」，X は「モデル行列 (*model matrix*)」もしくは「計画行列 (*design matrix*)」で列ベクトル

$x_0, x_1, \ldots, x_p$(決定項ベクトル) を持つ．ごく普通には x_0

は「切片項 (intercept)」を定義する，全て 1 からなる列ベクトルである．

Examples

形式的な仕様を与える前に，幾つかの例をあげると分かりやすいであろう．

数値変数 y, x, x0, x1, x2, . . . , 行列 X, 因子 A, B, C, . . . を考える．以下の左側の公式は，右側に与えられた統計モデル[1] を特定する．

`y ~ x`
`y ~ 1 + x` 両者はともに y の x への単純な線形回帰モデルを意味する．最初の式では切片項は明示されていず，第 2 式では明示されている．

`y ~ 0 + x`
`y ~ -1 + x`
`y ~ x - 1` 原点を通る (つまり切片項を持たない) y の x への単純な線形回帰モデル．

`log(y) ~ x1 + x2`
変換された変量の $x1$ and $x2$ への重回帰
$\log(y)$, (暗黙の切片項を持つ)．

[1] 訳注：実験計画は最初 R.A. Fisher により農業実験用に開発され，その歴史を反映するいくつかの用語を持つ．全区画 (*whole plot*) とは基本的栽培条件が同じになるような一区切りの実験用圃場であり，特定の栽培条件 (水準) を変えて栽培するために，それをいくつかの副区画 (*subplot*) に分割する．これを分割区画実験 (*split plot*) と称する．各副区画にどの水準を割り当てるかを無作為に決めるやり方が乱塊法 (*randomized block*) である．

```
y ~ poly(x,2)
y ~ 1 + x + I(x^2)
```
y の x への 2 次の多項式回帰．前者は基底関数として直交多項式を用い，後者は単なる巾乗式を用いる．

```
y ~ X + poly(x,2)
```
行列 X と x の 2 次多項式からなる計画行列を持つ y の重回帰．

`y ~ A` A で定まるクラスを持つ y の 1 元配置の分散分析．

`y ~ A + x` A で定まるクラスと，共変量 x を持つ y の 1 元配置の共分散分析．

```
y ~ A*B
y ~ A + B + A:B
y ~ B %in% A
y ~ A/B
```
2 つの因子 A と B を持つ y の 2 元配置の非加法的 (non-additive) モデル．最初の 2 つは同じ crossed classification を，残りの 2 つは同じ nested classification を指定する．抽象的には，これら 4 つは同じモデル空間を特定する．

```
y ~ (A + B + C)^2
y ~ A*B*C - A:B:C
```
主効果と 2 次交互作用のみを持つ 3 因子実験．両者は同じモデルを指定する．

```
y ~ A * x
y ~ A/x
y ~ A/(1 + x) - 1
```
\$A\$ の水準内で y を x に回帰する様々な単純線形回帰モデルで，異なったコーディングを持つ．最後の形は A に含まれる水準と同じ数の，異なった切片と傾きの推定値を陽に与える．\cr

```
y ~ A*B + Error(C)
```
2 つの処理因子 \$A\$ と \$B\$ と，因子 \$C\$ が定める誤差層 (error strata) を持つ実験．例えば，因子 \$C\$ から定まる全区画 (従って同じく副区画) を持つ分割区画実験．

演算子 ~ は R の「モデル式 (*model formula*)」を定義する．通常の線形モデルに対する形式は

response ~ op_1 term_1 op_2 term_2 op_3 term_3 ...

となる．ここで

response はベクトルか行列 (もしくは結果がベクトルか行列になる評価) で反応 (目的) 変数を定義する．

op_i は演算子で + か - である．ある項をモデルに含むか除外するかを表す (前者はオプションである)．

term_i は
- ベクトルか行列表現，もしくは 1，
- 因子，もしくは
- 「式演算子 (*formula operators*)」で結ばれた因子，ベクトル，または行列からなる「式表現 (*formula expression*)」．

全ての場合で，各項はモデル行列に加えられるか，除外される列ベクトルの集まりを定義する．1 は切片列ベクトルで，既定では明示的に除外されない限りモデル行列に含められる．

「式演算子 (*formula operators*)」は Glim や Genstat のようなプログラムで用いられる Wilkinson と Rogers の表記法と効果において似通っている．一つの避けられない変更は R ではピリオッドが意味のある名前用の文字であるために，演算子 ‘.’ が ‘:’ になることである．以下の表に演算子の表記が要約されている (Chambers & Hastie, 1992, p.29 に基づく)．

`Y ~ M` — Y は M と同様のモデル．

`M_1 + M_2` — *M_1* と *M_2* を含める．

`M_1 - M_2` — *M_1* を含め，*M_2* の項は除外せよ．

`M_1 : M_2` — *M_1* と *M_2* のテンソル積．もし両者が因子なら "サブクラス因子"．

`M_1 %in% M_2` — `M_1:M_2` に似るがコーディングが異なる．

`M_1 * M_2` — `M_1 + M_2 + M_1:M_2`.

`M_1 / M_2` — `M_1 + M_2 %in% M_1`.

`M^n` — M の全ての項と，n 次までの “交互作用”．

`I(M)` — M を孤立させる．M の中では，全ての演算子はその通常の数値演算での意味を持ち，その項はモデル行列に登場する．

関数の引数を普通囲む括弧の内部では，全ての演算子は通常の数値計算における意味を持つことを注意しよう．関数 `I()` は恒等関数で，モデル式中の項が数値演算子を用いて定義されることを許すためだけに使われる．

特にモデル式は「モデル行列の列 (*columns of the model matrix*)」を特定し，パラメータの指定は暗黙裡に行われることを注意しよう．

11.1.1 コントラスト

モデル式が如何にモデル行列の列を指定するか少なくとも多少の知識をもっておく必要がある．これはもし我々が連続変量ならば，各々がモデル行列の 1 つの列を提供するから，簡単である (そして切片はもしモデルに取り込まれれば全て 1 からなる列になる)．

k-個の水準を持つ因子 `A` に付いてはどうだろう? 答えは順序が付いた因子と，付かない因子では異なる．「順序無し (*unordered*)」因子では，$k-1$ 個の列が因子の第 2, . . . , 第 k 番目の水準の指標として生成される．(したがって暗黙のパラメータ化は各水準での反応を，最初のそれと比較対照させることにある．) 「順序付き (*ordered*)」因子では，$k-1$ 個の列は

集合 $1, \ldots, k$ 上の直交多項式であり，定数項は除かれる．

回答は既に複雑であるが，しかしながらこれだけではない．最初に，もし因子項を含むモデルに切片項が省略されていると，最初のそうした項が k 個の列に符号化され，全ての水準に対する指標を与える．第 2 に，全ての動作は `contrasts` に対する `options` の設定次第で変化する．R における既定の動作は次のようになっている．

```
options(contrasts = c("contr.treatment", "contr.poly"))
```

これを述べた主要な理由は，順序無し因子に対しては R と S は異なった既定動作を持つからである．S は Helmert コントラストを用いる．したがって，もし自分の結果を S-Plus を用いた本や論文のそれらと比較する必要があれば，次のようにしなければならないだろう．

```
options(contrasts = c("contr.helmert", "contr.poly"))
```

この違いは熟慮の上のものである．(R の既定である) 処理によるコントラストは初心者にはより解釈がしやすいであろうから．

これでまだ全部というわけではない．なぜなら，使用されるコントラストの用い方は，モデル中の各項に対し関数 `contrasts` と `c` を使って設定できるから．

交互作用項については未だ考えていなかった：これらは要素の項に対して導入された列の積を生成する．

詳細は複雑であるが，周辺項が保たれる限り，R のモデル公式は普通熟達の統計家が期待するであろうようなモデルを作り出す．例えば，対応する主効果が無い交互作用モデルは，驚くべき当ては め結果を導き，専門家のためだけのものである．

11.2 線形モデル

通常の多重モデルを当てはめるための基本的関数は `lm()` であり，その呼び出しの簡易版は次のようである：

> `> fitted.model <- lm(formula, data = data.frame)`

例えば

```
> fm2 <- lm(y ~ x1 + x2, data = production)
```

は y の x_1 と x_2 による (切片項を暗黙のうちに含む) 重回帰モデルを当てはめる．

重要だがテクニカルなオプションであるパラメータ `data=production` は，モデルを構成するのに必要な全ての変数は，先ず 「`production` データフレーム」から取られることを指示する．「これはデータフレーム `production` が既に検索リストに付け加えられているか否かによらない．」

11.3 モデル情報を取り出す総称的関数

`lm()` の返り値は当てはめられたモデルのオブジェクトである．技術的にはクラス`"lm"` の返り値であるリストである．当てはめられたモデルに関する情報は，その後でクラス `"lm"` のオブジェクトに方向づけられた総称的 (generic) な関数により，表示，抽出，描画等を行なうことが出来る．こうした関数としては次のようなものがある：

```
add1    coef      effects   kappa    predict   residuals
alias   deviance  family    labels   print     step
anova   drop1     formula   plot     proj      summary
```

もっとも普通に使われるものは以下で簡単に説明される．

`anova(`*`object_1`*`, `*`object_2`*`)`
: サブモデルと外部モデルを比較し，分散分析表を生成．

`coefficients(`*`object`*`)`
: 回帰係数 (行列) を抽出．

  省略形：`coef(`*`object`*`)`.

`deviance(`*`object`*`)`
: 必要なら重み付けられた，残差平方和．

`formula(`*object*`)`
: モデル式を抽出．

`plot(`*object*`)`
: 4 種類のプロットを生成．残差，当てはめ値，そして幾つかの診断．

`predict(`*object*`, 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．値は *data.frame* 中の非ランダムな変量に対応する予測値のベクトルもしくは行列．

`print(`*object*`)`
: オブジェクトの簡略版を表示．最もしばしば暗黙の内に使われる．

`residuals(`*object*`)`
: 適当に重み付けられた残差 (の行列) を抽出．
: 省略形：`resid(`*object*`)`.

`step(`*object*`)`
: 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．段階的探索で見付かった最大の AIC(赤池の情報量規準) 値を持つモデルが返される．

`summary(`*object*`)`
: 回帰分析の結果の完全な要約を表示．

11.4 分散分析とモデル比較

モデル当てはめ関数 `aov(formula, data=`*data.frame*`)` は，最も単純なレベルでは `lm()` と極めて似通った操作を行ない，表 Section 11.3 [モデル情報を取り出す総称的関数], page 53 にあげたほとんどの総称的な関数が使える．

`aov()` は更に分割区画実験や balanced incomplete block designs with recovery of inter-block information といった多重の誤差層を持つモデルの解析も行なえることを注意しよう．モデル式

```
response ~ mean.formula + Error(strata.formula)
```

は *strata.formula* で定義される誤差層を持つ多重誤差層実験を指定する．因子の水準の級内，級間という 2 層実験を定義するといういう最も簡単な場合，strata.formula は単に因子である．

たとえば，次の例のモデル式

```
> fm <- aov(yield ~ v + n*p*k + Error(farms/blocks), data=farm.data)
```

は典型的に平均モデル `v + n*p*k` と 3 つの誤差層，つまり「圃場間誤差」,「圃場内誤差」，そして「ブロック内誤差」を持つ実験を記述するのに用いられる．

11.4.1 ANOVA 表

分散分析表は当てはめられたモデルの系列に対するものであることを，同様に注意しよう．表示された自乗和は，モデル中の「その項」を系列中の「その場所」に含めたことに由来する残差平方和の減少分である．したがって，直交実験においてのみ，項を含める順序が非本質的になる．

多層実験に対する手順は，先ず応答を誤差層の上に射影 (再び順番に) し，それから平均モデルを各射影に当てはめる．これ以上の詳細については Chambers & Hastie (1992) を見よ．

既定値である完全な分散分析表に対するより柔軟性のある代替物は，2 つもしくはそれ以上のモデルを `anova()` 関数を使って比較することである．

```
> anova(fitted.model.1, fitted.model.2, ...)
```

結果は，すると順番に当てはめられた当てはめモデル間の差を示す分散分析表になる．比較されているモデルは，勿論普通階層的な系列であろう．これは既定値と異なる情報を与えるわけではなく，むしろ把握と制御をより簡単にするのである．

11.5 当てはめモデルの更新

update() 関数は多く，既に当てはめられたモデルに，普通ごくわずかの項を加えたり除いたりして得られるモデルの当てはめを容易に行なうために使われる．その形式は次のようである．

```
> new.model <- update(old.model, new.formula)
```

new.formula 中ではピリオッド '.' だけからなる特別な名前を「元のモデル式中の対応する部分」を表すのに使うことが出来る．例えば

```
> fm05 <- lm(y ~ x1 + x2 + x3 + x4 + x5, data = production)
> fm6  <- update(fm05, . ~ . + x6)
> smf6 <- update(fm6, sqrt(.) ~ .)
```

は，データフレーム (仮定では)production からのデータを用いた 5 変量の重回帰の当てはめを行ない，次に 6 番目の説明変数を取り込んで当てはめ，最後に目的変数に平方根変換を行なった変形モデルを当てはめている．

もし data= 引数が，最初のモデル当てはめ関数に指定されると，この情報は当てはめられたモデルオブジェクトを通じて update() やその仲間に伝えられることを，特に注意しよう．

名前 '.' はまた別の文脈でも使うことが出来るが，少し異なった意味を持つ．例えば

```
> fmfull <- lm(y ~ . , data = production)
```

は目的変数 y と「データフレーム production 中の他の全ての説明変数」を用いたモデルの当てはめを行なうであろう．

モデルの拡大系列を探求するための他の関数には add1(), drop1() そして step() がある．これらの関数の名前はその意味への良いヒントとなるが，完全な詳細についてはヘルプ文章を見よ．

11.6 一般化線形モデル

一般化線形モデルは，線形モデルを，非正規分布を持つ応答変数 (目的変数) とその線形化変換の双方を，明解で直接的な方法で取り扱得るように拡張したものである．一つの一般化線形モデルは，次の一連の仮定を用いて記述できる：

- 関心のある応答変数 y と，刺激変数
  $x_1, x_2, \ldots,$ があり，説明変数の値は目的変数の分布に影響を持つ．
- 刺激変数は y の分布に「ただ 1 つの線形関数を通じてのみ」影響を及ぼす．この線形関数は「線形予測量 (*linear predictor*)」と呼ばれ，通常

$$\eta = \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p,$$

  と書かれる．従って，x_i は $\beta_i = 0$ の時，そしてその時のみ，y の分布に影響を及ぼさない．
- y の分布は次の形

$$f_Y(y;\mu,\varphi) = \exp\left[\frac{A}{\varphi}\left\{y\lambda(\mu) - \gamma\left(\lambda(\mu)\right)\right\} + \tau(y,\varphi)\right]$$

を持つ．ここで φ は「スケールパラメータ (*scale parameter*)」(既知の場合もある) で，全ての観測値に対し定数である．A は予め与えられた重みで，既知と仮定されるが，観測値ごとに異なるかも知れない．μ は y の平均である．従って y の分布はその平均，そしておそらくスケールパラメータにも依存して決まる．

•

平均 μ は線形予測量の可逆で滑らかな関数

$$\mu = m(\eta), \qquad \eta = m^{-1}(\mu) = \ell(\mu)$$

であり，この可逆関数 $\ell()$ は「連結関数 (*link function*)」と呼ばれる．

これらの仮定は，実際の統計解析で有用な幅広いクラスのモデルを包含するのに十分なくらい緩いが，推定と推測の統一した理論を (少なくとも近似的には) 展開するのに十分なほど狭い．詳細については，この話題を扱った McCullagh & Nelder (1989) や Dobson (1990) といった参考文献を見ることをすすめたい．

11.6.1 ファミリ

R が提供する機能で扱える一般化線形モデルのクラスは，応答分布 (応答変数の分布) が正規分布，2 項分布，ポアッソン分布，逆正規分布，そしてガンマ分布の場合を含む．また応答分布がはっきりと特定されていない場合に対する疑似尤度モデルを含む．最後の場合には「分散関数」が平均の関数として特定されている必要があり，それ以外ではこの関数は応答分布から決まる．

各応答分布には，平均と線形予測量を結ぶ様々な連結関数を対応させることが出来る．自動的に使用可能なものを次の表にまとめた．

ファミリ名	連結関数
`binomial`	`logit`, `probit`, `cloglog`
`gaussian`	`identity`
`Gamma`	`identity`, `inverse`, `log`
`inverse.gaussian`	`1/mu^2`
`poisson`	`identity`, `log`, `sqrt`
`quasi`	`logit`, `probit`, `cloglog`, `identity`, `inverse`, `log`, `1/mu^2`, `sqrt`

モデル作成に必要な，応答分布，連結関数，そして，その他の様々な情報の組み合わせは，一般化線形モデルの「ファミリ (*family*)」と呼ばれる．

11.6.2 `glm()` 関数

応答分布は「ただ 1 つ」の線形変換を通じてのみ刺激変数に依存しているので，一般化モデルの線形部分を特定するには，線形モデルに対して使われたのと同じメカニズムを使うことができる．ファミリは別の方法で特定されねばならない．

一般化線形モデルを当てはめる R の関数は `glm()` で，次の形を用いる：

```
> fitted.model <- glm(formula, family=family.generator, data=data.frame)
```

唯一の新しい特徴はファミリを記述する道具である「ファミリ生成子 (*family.generator*)」である．これは，一緒になってモデルと推測の過程を定義し制御する関数と表現のリストを生成する関数

の名前である．これは最初一見すると，少々複雑に見えるかも知れないが，その使用法は極めて簡単である．

標準的に提供されているファミリ生成子の名前は表 Section 11.6.1 [ファミリ], page 56 中の「ファミリ名」の欄に与えられている．もし連結関数が選択できるなら，連結の名前を括弧にいれて，パラメータとしてファミリ名に加えることができる．「疑似 (quasi)」ファミリの場合には，分散関数を同様に特定することができる．

いくつかの例をあげると仕組みが分かりやすいであろう．

正規 (gaussian) ファミリ

次のような呼出し

```
> fm <- glm(y ~ x1 + x2, family = gaussian, data = sales)
```

は

```
> fm <- lm(y ~ x1+x2, data=sales)
```

と同じ結果を与えるが，ひどく効率が悪くなる．連結関数の選択が自動的に正規ファミリを選ぶのでは無いこと，したがっていかなるパラメータも許されないことを注意しよう．もし問題が非標準的な連結を持つ正規ファミリを必要とするならば，後で分かるように，それは通常疑似 (quasi) ファミリによって達成される．

2 項 (binomial) ファミリ

Silvey (1970) にある簡単な人工的な例を考えよう．

エーゲ海諸島のカリトス島の男性住民は，年齢とともに進行する，遺伝的な目の病気にかかる．さまざまな年齢の島の男性住民の盲目度が調査され，記録された．データは以下の示されている：

年齢:	20	35	45	55	70
No. 被検者数:	50	50	50	50	50
No. 盲目者数:	6	17	26	37	44

我々の考える問題は，このデータにロジスティックモデルとプロビットモデルを当てはめ，各モデルに対する LD50 値，つまり男性住民が盲目になる可能性が 50% を越える年齢を推定することである．

y を年齢 x での盲目者数，n を被検者数とすると，両方のモデルは

$$y \sim \mathrm{B}(n, F(\beta_0 + \beta_1 x))$$

の形になる．ここでプロビットモデルでは

$F(z) = \Phi(z)$ 標準正規分布であり，ロジットモデル (既定) では

$F(z) = e^z/(1+e^z)$ である．双方のモデルで LD50 値は

$$\mathrm{LD50} = -\beta_0/\beta_1$$

となる，つまり分布関数の引数がゼロとなるような点である．

最初のステップはデータをデータフレームとしてセットすることである

```
> kalythos <- data.frame(x = c(20,35,45,55,70), n = rep(50,5),
                         y = c(6,17,26,37,44))
```

glm() を用いて 2 項モデルを当てはめるには応答変数として 2 つの可能性がある：

- もし応答が「ベクトル」ならば，それは「バイナリ (2 値)」データを含んでいると仮定され，結局 0/1 ベクトルでなければならない．
- もし応答が「2 列の行列」ならば，最初の列は試行における成功の数，第 2 列は失敗の数を含んでいると仮定される．

ここでは，後者の場合を考えるので，データフレームに行列を加える：

```
> kalythos$Ymat <- cbind(kalythos$y, kalythos$n - kalythos$y)
```

モデルを当てはめるには次のようにする．

```
> fmp <- glm(Ymat ~ x, family = binomial(link=probit), data = kalythos)
> fml <- glm(Ymat ~ x, family = binomial, data = kalythos)
```

ロジット連結が既定値であるので，第 2 の呼出しではパラメータを省略しても良い．各当てはめの結果を見るには次のようにする．

```
> summary(fmp)
> summary(fml)
```

ともに (うますぎるほど) 良い当てはめを示す．LD50 値の推定値を見付けるには次のような簡単な関数を使う：

```
> ld50 <- function(b) -b[1]/b[2]
> ldp <- ld50(coef(fmp)); ldl <- ld50(coef(fml)); c(ldp, ldl)
```

このデータから得られる実際の推定値は，それぞれ 43.663 歳と 43.601 歳である．

ポアッソンモデル

ポアッソンモデルに対する既定の連結は `log` であり，現実にはこのファミリの主な利用は頻度データにポアッソン対数線形モデルを代用物として当てはめることにあり，その実際の分布はしばしば多項分布になる．これは重要な大きな主題であり，ここではこれ以上議論することは無い．それは全体としての非正規一般化モデルの利用の主要な部分を占めさえする．

時折，真にポアッソン分布に従うデータが現実に登場し，過去にはそれは対数か平方根変換した後で正規データとして解析されることがしばしばあった．後者に対する上品な代用品として，次の例のように，ポアッソン一般化線形モデルを当てはめることができる：

```
> fmod <- glm(y ~ A + B + x, family = poisson(link=sqrt),
              data = worm.counts)
```

疑似尤度モデル

すべてのファミリで，応答の分散は平均に依存し，スケールパラメータ (尺度母数) が乗算因子として掛かって来る．平均への依存の仕方は応答変数の分布の特性である；たとえばポアッソン分布では $\mathrm{Var}[y] = \mu$ となる．

疑似尤度モデルによる推定と推測では，応答分布の正確な形は特定されず，むしろ連結関数と分散関数の平均への依存の形式だけが指定される．疑似尤度推定は，形式的には正規分布に対するのと同一のテクニックを使うので，このファミリは非標準的な連結関数や分散関数を持つ正規モデルを当てはめる手法を，たまたま提供する．

たとえば，次の非線形回帰

$$y = \frac{\theta_1 z_1}{z_2 - \theta_2} + e$$

を考える．これは次のようにも表せる

$$y = \frac{1}{\beta_1 x_1 + \beta_2 x_2} + e.$$

ここで

$x_1 = z_2/z_1$, $x_2 = -1/x_1$, $\beta_1 = 1/\theta_1$ そして $\beta_2 = \theta_2/\theta_1$ である．適当なデータフレームがセットされていると仮定すると，この非線形回帰は次のように当てはめることができるであろう

```
> nlfit <- glm(y ~ x1 + x2 - 1,
               family = quasi(link=inverse, variance=constant),
               data = biochem)
```

これ以上の情報に付いては，必要に応じてマニュアルとヘルプ文章を参照してほしい．

11.7 非線形最小自乗法と最尤モデル

ある種の非線形モデルへの当てはめを一般化線形モデル (`glm()`) を用いて行うことができる．しかし多くの場合，非線形曲線の当てはめ問題は非線形最適化問題として扱うべきである．R の非線形最適化ルーチンは `nlm()` であり，これは S-Plus の `ms()` と `nlmin()` に取って代わるものである．我々はある当てはめの善し悪しをはかる指標を最小化するパラメータを求めたく，`nlm()` は様々なパラメータ値を繰り返し適用することによりこれを行う．例えば線形モデルとは異なり，過程が満足できる推定値に収束する保証は無い．`nlm()` は最初に試みるべき初期値を必要とし，収束はこの初期値の推測の善し悪しに本質的に依存する．

11.7.1 最小自乗法

非線形モデルを当てはめる 1 つの方法は自乗誤差和 (SSE) もしくは残差を最小化することである．この方法は観測誤差が正規分布に概略従う時に意味を持つ．

Bates & Watts (1988), 51 ページに 1 つの例がある．データは次の通りである．

```
> x <- c(0.02, 0.02, 0.06, 0.06, 0.11, 0.11, 0.22, 0.22, 0.56, 0.56,
         1.10, 1.10)
> y <- c(76, 47, 97, 107, 123, 139, 159, 152, 191, 201, 207, 200)
```

当てはめられるモデルは次の通りである．

```
> fn <- function(p) sum((y - (p[1] * x)/(p[2] + x))^2)
```

当てはめを行うためにはパラメータの初期推定が必要になる．実用的な初期値を見出す 1 つの方法はデータをプロットし，あるパラメータ値を推測し，この値を用いてモデル曲線を上書きしてみることである．

```
> plot(x, y)
> xfit <- seq(.02, 1.1, .05)
> yfit <- 200 * xfit/(0.1 + xfit)
> lines(spline(xfit, yfit))
```

もっとましなものもあるかも知れないが，この 200 と 0.1 という初期値は満足すべきものと思われる．ここで当てはめを行おう．

```
> out <- nlm(fn, p = c(200, 0.1), hessian = TRUE)
```

当てはめの後では，`out$minimum` は SSE であり，そして `out$estimates` はパラメータの最小自乗推定量である．推定量の近似標準誤差 (standard errors, SE) を求めるには次のようにする：

```
> sqrt(diag(2*out$minimum/(length(y) - 2) * solve(out$hessian)))
```

上の行の中の 2 はパラメータの数を表す．1 つの 95%信頼区間はパラメータ推定値 ± 1.96 SE である．新しいプロットに最小自乗フィットを重ね書きしてみることができる．

```
> plot(x, y)
> xfit <- seq(.02, 1.1, .05)
> yfit <- 212.68384222 * xfit/(0.06412146 + xfit)
> lines(spline(xfit, yfit))
```

標準パッケージ **nls** は最小自乗法による非線形モデルの当てはめに対するもっと広範囲の手段を提供する．上で当てはめたモデルは Michaelis-Menten モデルであり，したがって次のようにすることができる．

```
> df <- data.frame(x=x, y=y)
> fit <- nls(y ~ SSmicmen(x, Vm, K), df)
> fit
Nonlinear regression model
  model:  y ~ SSmicmen(x, Vm, K)
   data:  df
          Vm           K
212.68370711   0.06412123
 residual sum-of-squares:  1195.449
> summary(fit)

Formula: y ~ SSmicmen(x, Vm, K)

Parameters:
    Estimate Std. Error t value Pr(>|t|)
Vm 2.127e+02  6.947e+00  30.615 3.24e-11
K  6.412e-02  8.281e-03   7.743 1.57e-05

Residual standard error: 10.93 on 10 degrees of freedom

Correlation of Parameter Estimates:
      Vm
K 0.7651
```

11.7.2 最尤法

最小自乗法は誤差が正規分布に従わないときでも使える非線形当てはめの手法である．この方法は対数尤度を最大化，同じことだが対数尤度の負値を最小化する，パラメータ値を求める．Dobson (1990), 108–111 ページにある例をここで紹介しよう．この例は投薬量に対する反応データにロジスティックモデルを当てはめるものであり，明らかに `glm()` を用いても解析できる．

```
> x <- c(1.6907, 1.7242, 1.7552, 1.7842, 1.8113,
         1.8369, 1.8610, 1.8839)
> y <- c( 6, 13, 18, 28, 52, 53, 61, 60)
> n <- c(59, 60, 62, 56, 63, 59, 62, 60)
```

最小化すべき負対数尤度は以下のようになる：

```
> fn <- function(p)
   sum( - (y*(p[1]+p[2]*x) - n*log(1+exp(p[1]+p[2]*x))
           + log(choose(n, y)) ))
```

分別ある初期値を丁寧に選び当てはめを行う：

```
> out <- nlm(fn, p = c(-50,20), hessian = TRUE)
```

当てはめの後では，out$minimum は負対数尤度，そして out$estimates はパラメータの最尤推定量である．推定量の近似 SE を求めるには次のようにする：

```
> sqrt(diag(solve(out$hessian)))
```

上の行の中の 2 はパラメータの数を表す．1 つの 95% 信頼区間はパラメータ推定値 $\pm$ 1.96 SE と取ることができる．

11.8 幾つかの非標準モデル

この章の最後に，R が提供する他の特殊な回帰とデータ解析用の機能を幾つか簡略に紹介する．

- **Mixed models.** ユーザ提供のパッケージ **nlme** は，係数の一部分がランダム効果に対応する，線形および非線形混合効果モデルに対する関数 lme() と nlme() を提供する．これらの関数はモデルを特定するために公式を駆使する．
- 局所近似回帰． 関数 loess() は，局所的な重みを用いた回帰を用い，ノンパラメトリックな回帰の当てはめを行う．こうした回帰はややこしいデータの傾向を際だたせたり，大規模データに対する何らかの知見を得る目的でデータの縮約を行うために有効である．

  関数 loess は射影追跡回帰 (projection pursuit regression) 用のコードとともに，標準パッケージ **modreg** にある．
- 頑健な回帰．データ中の極端な外れ値の影響に対し抵抗力を持たせるように回帰モデルを当てはめるための関数がいくつかある．同名の標準パッケージ **robust** 中の関数 lqs は高度に頑健な当てはめに対する最先端のアルゴリズムを提供する．頑健さは劣るが統計的により効率的な方法がユーザ提供のパッケージ中にある．例えばパッケージ **MASS** 中の関数 rlm が一例である．
- 加法モデル．この手法は，決定変数の滑らかな加法的関数 (普通，各決定変数にたいし 1 つずつ) から回帰関数をつくり出すことを目指している． ユーザ提供の R のパッケージ **acepack** 中の関数 avas() や ace()，ace そしてパッケージ **mda** 中の関数 bruto と mars がこうした手法の幾つかの例を与える．
- 木構造に基づくモデル．木 (ツリー) に基づくモデルは，予測や解釈のために正確な大局的線形モデルを求めるというよりは，最終的にデータを，グループ内ではできる限り等質的，グループ間ではできる限り異質的になるように，決定変数のクリティカルな値で再帰的にデータを分岐させることを目指している．その結果は，しばしば，他のデータ解析の手法では得にくいような洞察を与える．

  モデルは再び通常の線形モデルの形式で指定される．モデル当てはめ関数は tree() であるが，plot() や text() といった多くの他の関数は，木に基づくモデルの当てはめの結果をグラフィカルに表示するのにうまく適合している．

  木構造に基づくモデルはユーザ提供のパッケージ **rpart** と **tree** を用いて R で使用できる．

12 グラフィックスの取扱い

作図機能は重要なで極めて多岐にわたる R 環境の要素である．この機能を用いて，多彩な統計グラフを表示したり，また全く新しいタイプのグラフを構成することが出来る．

作図機能は対話的・非対話的なモードの双方で利用できるが，多くの場合対話的な利用の方がより生産的である．対話的な利用は，起動時に R が対話的なグラフ表示のための特殊な「グラフィックスウィンドウ」を開く作図「デバイスドライバ (`device driver`)」を初期化するので，また簡単でもある．

これは自動的になされるが，使用されている命令が Unix 上では `X11()`，Windows では `Windows()`，MacOS 8/9 では `macintosh()` であることを知っておくと便利である．

一度デバイスドライバが稼働し始めると R の作図命令は，様々なグラフ表示を生成したり，全く新しい種類の表示を作るために利用できる．

作図命令は 3 つの基本的なグループに分けられる．

- 「高水準」作図関数は，作図デバイスに新しい図形を表示し，これは普通軸・ラベル・表題などを伴う．
- 「低水準」作図関数は，既存の図に余分の点・直線・ラベルなどの付加情報を追加する．
- 「対話的」作図関数は，マウスなどの位置指示デバイスを使って，対話的に既存の図に情報を追加したり，抜きだしたりすることを可能にする．

更に R は使用者が図のカスタマイズを処理するために使うことができる「作図パラメータ」のリストを保持している．

12.1 高水準プロット命令

高水準作図関数は，関数に引数として渡されたデータの，完全なプロットを生成するように設計されている．もしそれが適当なら，軸・ラベルそして表題が自動的 (別の事を指示しない限り) に生成される．高水準作図命令は常に新しい図を開始し，必要ならば現在の図を消しさる．

12.1.1 `plot()` 関数

R において最も頻繁に利用される作図関数の 1 つが `plot()` 関数である．これは「総称的 (*generic*) な」関数である．生成されるプロットのタイプは第 1 引数のタイプや「クラス」に依存する．

`plot(`*x*`, `*y*`)`
`plot(`*xy*`)` x と y がベクトルなら，`plot(`*x*`,`*y*`)` は y に対する x の散布図を作成する．同じ効果は，2 つの成分 x と y を含むリストや，2 列の行列のいずれかを，単独の引数として与える (第 2 の形式) ことによっても得られる．

`plot(`*x*`)` もし x が時系列なら，時系列プロットを生成し，x が数値ベクトルなら，ベクトル中の数値を，そのベクトルの添字に対してプロットし，またもし x が複素ベクトルなら，ベクトル成分の実部に対する虚部のプロットを生成する．

`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．最初の形式は y の棒グラフを，第 2 の形式は f の各水準に対する y の箱型図 (box plot) を生成する．

plot(*df*)
plot(~ *expr*)
plot(*y* ~ *expr*)
: *df* はデータフレーム，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．最初の 2 つの形式は，データフレーム中の変量 (第 1 形式)，または名前が与えられたオブジェクト (第 2 形式)，の分布関数プロット (distributional plot) を生成する．第 3 の形式は expr に名前が与えられた全てのオブジェクトに対して y をプロットする．

12.1.2 多変量データを表示する

R は多変量データを表現する 2 つの大変便利な関数を提供する．X が数値行列またはデータフレームならば，命令

```
> pairs(X)
```

は X の列で定義される変量の対毎の散布図の行列 (pairwise scatterplot matrix) を生成する，つまり X の全ての列が他の全ての列に対してプロットされ，得られた $n(n-1)$ 個のプロットは行列型に配置され，その作図スケールは行列の行と列に対して共通となる．

3 つもしくは 4 つの変量が含まれるときは「共変量プロット (*coplot*)」がより分かりやすいであろう．もし a と b が数値ベクトルで，c が (全てが同じ長さの) 数値ベクトル，もしくは因子オブジェクトなら，命令

```
> coplot(a ~ b | c)
```

は c の与えられた値における b に対する a の散布図を複数生成する．もし c が因子なら，これは単に c の全ての水準毎に，a が b に対してプロットされることを意味する．c が数値の場合，それはいくつかの「条件を与える区間」に分割され，それぞれの区間毎に，その区間に含まれる c の値に対して a が b に対してプロットされる．区間の数と位置は coplot() に対する given.values= 引数により制御することができる— 関数 co.intervals() は区間の選択に便利である．また 2 つの「与えられた」変量を次のような命令

```
> coplot(a ~ b | c + d)
```

で使うこともでき，これは c と d の条件付け区間をあわせたものにおける a の b に対する散布図を生成する．

coplot() 関数と pairs() 関数はともに panel= 引数をもち，それぞれのパネルに現れるプロットのタイプをカスタマイズすることが出来る．既定では散布図を生成する points()であるが，panel= の値として x と y の 2 つのベクトルの他の低水準作図関数を与えることにより，希望するいかなるタイプのプロットも生成することができる．共変量プロットに対する便利なパネル関数の例は panel.smooth() である．

12.1.3 グラフィックスの表示

他の高水準作図関数は異なったタイプのプロットを生成する．いくつかの例は：

qqnorm(x)
qqline(x)
qqplot(x, y)
: 分布を比較するプロット．最初の形式は x に対する期待正規ランクスコアをプロットし (正規スコアプロット)，第 2 の形式はそうしたプロットに，データの上四分位数と下四分

位数を結ぶ直線を描くことにより，そのプロットに直線を追加する．第 3 の形式は，それぞれの分布を比較するために x の確率点に対する y の確率点をプロットする．

`hist(x)`
`hist(x, nclass=`*n*`)`
`hist(x, breaks=`*b*`, ...)`
: 数値ベクトル x のヒストグラムを生成する．通常は適切な階級数が選ばれるが `nclass=` 引数で望みの階級数を指定することが出来る．もしくは `breaks=` 引数により，区切り点 (breakpoints) を正確に指定することが出来る．`prob=T` 引数が与えられたとき，柱は度数ではなく相対頻度を表す．

`dotchart(x, ...)`
: x におけるデータの点グラフ (dotchart) を構成する．点グラフにおいては y-軸は x におけるデータのラベル付けを与え，x-軸はその値を与える．例えばこのグラフは，特定の範囲にある全てのデータ項目を容易に視覚的に選択することを可能にする．

`image(x, y, z, ...)`
`contour(x, y, z, ...)`
`persp(x, y, z, ...)`
: 3 変量のプロット．`image` プロットは `z` の値を表すため，異なった色を用いた矩形の格子を描く．`contour` プロットは `z` の値を表すため，等高線を描く．`persp` プロットは 3 次元的に曲面を描く．

12.1.4 高水準プロット関数への引数

高水準作図関数に渡すことができる以下のようないくつかの引数がある：

`add=TRUE`
: その関数が低水準作図関数として動作するよう強制し，現在のプロットにプロットを上描きする (使える関数は限られる)．

`axes=FALSE`
: 軸の生成を抑制する — `axis()` 関数を用いて自分自身の設定した軸を追加するのに有用である．既定である `axes=T` は軸を含むことを意味する．

`log="x"`
`log="y"`
`log="xy"`
: x，y もしくは双方の軸を対数軸とする．これは多くのプロットに対し可能であるが，いつでも可能であるわけではない．

`type=`
: `type=` 引数は生成されるプロットのタイプを以下のように制御する：

  `type="p"` 個々の点を描く (既定)

  `type="l"` 線を描く

  `type="b"` 線で連結された点を描く (*both*)

  `type="o"` 点を描き，線で上描き

  `type="h"` 点からゼロ軸 (x 軸) まで鉛直線を描く (*high-density*)

  `type="s"`
  `type="S"` 階段関数プロット．最初の形式は鉛直方向の頂点が，第 2 形式では基部がその点を定義する．

`type="n"` 全くなにも描かない．しかし (既定では) 軸はやはり描かれ，座標軸がデータに基づいて設定される．引き続いて低水準作図関数を用いた作図をするのに最適．

`xlab=`*`string`*
`ylab=`*`string`*
: x 軸と y 軸の軸ラベル．既定のラベル (高水準作図関数の呼び出し時には一般にオブジェクト名) をかえるためにこれらの引数を使おう．

`main=`*`string`*
: 図表の上部に置かれる，大きなフォントで描かれる図の表題．

`sub=`*`string`*
: x-軸の直下に置かれる，より小さなフォントで描かれる副題．

12.2 低水準プロット命令

高水準作図関数は場合によって，希望する種類の図表を正確には生成しない．その場合低水準作図命令を用いて現在の図表に対して余分の情報 (点や線やテキストなど) を追加するために利用することができる．

いくつかのより便利な低水準作図関数は：

`points(x, y)`
`lines(x, y)`
: 現在の図表に点もしくは連結線を描く．`plot()` の `type=` 引数もまた，これらの関数に渡すことができる (既定では `points()` に対して `"p"` を，`lines()` に対して `"l"` を与える)．

`text(x, y, labels, ...)`
: `x, y` で与えられる点にテキストを置く．通常 `labels` は整数もしくは文字ベクトルであり，そのとき `labels[i]` は点 (`x[i]`，`y[i]`) に描かれる．既定では `1:length(x)`．

「注意」: この関数は通常

```
> plot(x, y, type="n"); text(x, y, names)
```

と引き続いて使われる．作図パラメータ `type="n"` はその点の表示を抑制するが，軸を構成する．そして，`text()` 関数は，その点に対して文字ベクトル `names` により特定される，特別な文字を提供する．

`abline(a, b)`
`abline(h=`*`y`*`)`
`abline(v=`*`x`*`)`
`abline(`*`lm.obj`*`)`
: 傾きが `b`，切片が `a` の直線を現在の図表に追加する．`h=y` は図表を横切る水平線の高さの y-座標を特定するのに，同様に `v=x` は垂直線の x-座標を指定するのに使うことができる．同様に $lm.obj$ は (モデル当てはめ関数の結果等の) 長さ 2 の `coefficients` 成分を持つリストであり，それらは順に引かれるべき直線の切片と傾きになる．

`polygon(x, y, ...)`
: (x, y) を順序の付いた頂点とする多角形を描き，(オプションで) 陰影線により影をつけたり，作図デバイスが塗りつぶしを許す場合には塗りつぶしを行う．

`legend(x, y, legend, ...)`

現在の図の指定された位置に凡例 (legend) を付加する．作図記号・線種・色などは，ベクトル `legend` で与えられた文字ベクトルのラベルにより識別される．少なくとも 1 つの他の，作図単位で表された値を持つ，引数 y(legendと同じ長さのベクトル) が以下のように与えられねばならない．

`legend( , fill=`*v*`)`
箱を塗りつぶす色

`legend( , col=`*v*`)`
点や線を描く色

`legend( , lty=`*v*`)`
線の種類

`legend( , lwd=`*v*`)`
線の幅

`legend( , pch=`*v*`)`
プロット用文字 (文字ベクトル)

`title(main, sub)`

表題 `main` を現在の図表の上部に大きな文字で描く，(オプションとして) 副題 `sub` をより小さな文字で基部に描く．

`axis(side, ...)`

現在の図表に，第 1 引数により与えられた側 (1 から 4，基部から時計回りで数える) に軸を追加する．他の引数は軸を図表の内側か外側に描くかどうか，目盛り位置，ラベルなどを制御する．`axes=F` 引数付きで `plot()` を呼んだ後で，好みの軸を加える場合に便利である．

低水準作図関数は通常新しい作図要素を置く位置を決めるために，位置情報を要求する (例えば x と y の座標)．座標は先立つ高水準作図関数により定義された「ユーザ定義座標」に対して与えられ，提供されたデータに基づいて選ばれる．

`x` と `y` 引数が要求されるところでは，`x` と `y` と名付けられた要素を持つリストである 1 つの引数を与えても同じく十分である．同様に 2 つの列からなる行列もまた正当な入力である．このような方法で `locator()`(後述) のような関数も図表における位置を対話的に指定するために使うことができる．

12.2.1 数式による注釈

プロットに数学記号や式を使うことが有用な場合がある．これは R で `text`, `mtext`, `axis` もしくは `title` のいずれかを使う際，文字列の代わりに「表現 (*expression*)」を指定することで可能になる．例えば以下のコードは 2 項確率関数に対する公式を書く：

```
> text(x, y, expression(paste(bgroup("(", atop(n, x), ")"),
                              p^x, q^{n-x})))
```

可能な全ての特徴を含むより多くの情報が R から次の命令を使うと得られる：

```
> help(plotmath)
> example(plotmath)
```

12.2.2 Hershey ベクトルフォント

`text` や `contour` 関数を使ってテキストを返すとき Hershey ベクトルフォントを指定できる．Hershey フォントを使う 3 つの理由がある：

- Hershey フォントは，特にテキストを回転したり小くする時に，より良い出力 (特に計算機画面上) を出せる．
- Hershey フォントは標準フォントには無いかも知れないある種の記号を提供する．特に，占星術用記号，地図用記号，そして天文学用記号である．
- Hershey フォントはキリル文字と日本語文字 (かなと漢字) を提供する．

Hershey 文字表を含むより多くの情報は R から次の命令で得ることができる：

```
> help(Hershey)
> example(Hershey)
> help(Japanese)
> example(Japanese)
```

12.3 グラフィックスの対話的操作

R は使用者がマウスを使って，図表から情報を除いたり追加したりする事を可能にする関数も用意している．最も簡単なものは `locator()` 関数である：

`locator(n, type)`
: ユーザが図表においてマウスの左ボタンで位置を選択するの待つ．これはユーザが `n`(既定値は 512) 点選ぶまでか，

  別のボタンを押す (Unix，Windows) までか，グラフィックウィンドウの外でクリックする (Mac) まで，

  続けられる．`locator()` は選択された位置を，2 つの要素 `x` と `y` のリストとして返す

`locator()` は通常引数を伴わずに呼ばれる．これは凡例やラベルなどのグラフ要素の位置を，グラフのどの位置に置くかあらかじめ計算するのが困難なときに，対話的に選択するために特に便利である．例えば外れ値の近くにいくつかの有益なテキストを置く場合，次の命令

```
> text(locator(1), "Outlier", adj=0)
```

は便利だろう．`locator()` は現在のデバイスがマウスをサポートしていない場合にも動作する．この場合はユーザが x と y 座標を入力することを促すプロンプトが現れる．

`identify(x, y, labels)`
: ユーザが (マウスの左ボタンを使って) 定義した x と y の点を，その近くに関連する `labels` の要素 (もしくは `labels` がなければ観測番号) を描くことにより，強調表示することを可能にする．

  別のボタンを押す (Unix，Windows) か，グラフィックウィンドウの外でクリックする (Mac) と，

  選択した点の観測番号を返す．

場合によっては我々は作図上の特定の「点」の位置を知るよりも，それを識別したい．例えばいくつかの興味ある観測値を作図表示から選択し，それらを何らかの方法で操作したい．2 つの数値ベクトル x と y のうちのいくつかの (x, y) 座標が与えられれば，我々は `identify()` 関数を以下のように利用することができるであろう：

```
> plot(x, y)
> identify(x, y)
```

identify() 関数はそれ自身では作図は行わないが，ユーザがマウスポインタを動かし，ある点の近くでマウスの左ボタンをクリックすることを許す．マウスポインタに最も近い点が，観測番号値を近くに表示する (つまり，x/y ベクトルにおける順番) ことにより強調される．また使用者は identify() に対して labels 引数を利用することにより，(例えばケース名など) 有益な文字列を強調表示として利用したり，plot=F 引数を与えて強調を全くしないようにすることもできる．

点の選択を終了する (上述) と，

identify() は選択された点の番号を返す．つまり使用者はこれらの添字をもともとの x と y のベクトルから選び出すために利用することができる．

12.4 グラフィックスパラメータを使う

グラフを作るとき，特に発表や出版の目的の場合，R は必ずしも要求された通りのものを生成しない．しかしながら「作図パラメータ」を使えば，表示のほとんど全ての見掛けをカスタマイズできる．R は中でも線分の形式・色・図表の配置，及びテキスト位置の整形等の様々なものを制御する数多くの作図パラメータのリストを維持する．全ての作図パラメータは (色を制御する ‘col’ などの) 名前と値 (例えば色番号) を持っている．

動作中の各デバイスのために，別々のグラフパラメータが保持されており，各デバイスは初期化の際既定値からなるパラメータのセットを持つ．2 つの方法で作図パラメータを設定することができる：現在のデバイスにアクセスする全ての作図関数に影響する恒久的なものと，1 つの作図関数だけに影響する一時的なもののどちらかである．

12.4.1 永続的変更：par() 関数

par() 関数は現在の作図デバイスに対する作図パラメータのリストにアクセスし，修正するために利用される．

par()
: 引数が無いと，現在のデバイスに対する全ての作図パラメータとそれらの値を返す．

par(c("col", "lty"))
: 文字ベクトル引数を伴うと，その名前の付いた作図パラメータのみを (再びリストとして) 返す．

par(col=4, lty=2)
: 名前付きの引数 (もしくは単一のリスト引数) により，その名前の作図パラメータの値を設定し，もとのパラメータの値をリストとして返す．

par() 関数で作図関数を指定することは，その後の全ての (現在のデバイスにおける) 作図関数の呼び出しが新しい値の影響を受けると言う意味で「恒久的に」パラメータの値を変更する．このように作図パラメータを設定することは，パラメータの「既定値」を設定することと考えられ，別の値が与えられるまでこの値が全ての作図関数で使われるであろう．

par() の呼び出しは「常に」，たとえ par() がある関数において呼び出されたときでさえ，作図パラメータの大局的な値に影響を与えることを注意せよ．このことはしばしば望ましい振る舞いではない — 通常我々はいくつかの作図パラメータを設定し，作図を行い，そしてもとの値を回復し，ユーザの R セッションに影響を与えないようにしたい．変更をする際に par() の結果を保存し，作図が完了したときに初期値に戻すことによって，初期値を回復させることができる．

```
> oldpar <- par(col=4, lty=2)
  ... 作図命令 ...
> par(oldpar)
```

12.4.2 一時的変更：グラフィックス関数への引数

作図パラメータは同様に，名前付きの引数として，(ほとんど) 全ての作図関数に渡すことができる．これは，変更が関数呼び出し中だけであることを除けば，par() 関数に引数を渡すのと同様の効果がある．例えば

```
> plot(x, y, pch="+")
```

はプラス記号を作図文字として使う散布図を生成するが，将来の作図に対する既定作図文字を変更しない．

12.5 グラフィックスパラメータのリスト

以下の節では普通に使用される多くの作図パラメータを詳しく述べる．R の par() 関数のヘルプドキュメントにはもっと簡潔な要約が提供されているが，これは少しより詳しい代替物を与える．

作図パラメータは次の形式で提示されるだろう：

name=value
: パラメータの効果の記述．name はパラメータ名，つまり par() や作図関数を呼ぶ際に使う引数の名前である．value はパラメータを設定する際に使用者が使うであろう典型的な値である．

12.5.1 グラフィックスの要素

R の作図は，点・線・テキスト，そして多角形 (塗りつぶされた領域) から構成されている．以下のような，どのように「グラフ要素」を描くかを制御する作図パラメータがある：

pch="+"
: 点を描くために用いられる文字．既定値は作図デバイスに応じて変わるが，通常は
‘◦’. である．作図文字として "." を用いると中心に点を置かれる以外は，作図された点は適切な位置より若干上もしくは下に現れる傾向がある．

pch=4
: pch が 0 から 18(18 を含む) の間に含まれる整数として与えられたとき，指定された作図記号が生成される．どんな記号かを知るためには次の命令を使う

```
> legend(locator(1), as.character(0:18), pch=0:18)
```

lty=2
: 線分の種類．他の線種が全ての作図デバイスで使えるわけではない (また作図デバイスにより振る舞いも異なる) が，しかし線種 1 は常に実線であり，線種 2 とそれ以上の値は点線もしくは波線，もしくはその双方を組み合わせたものである．

lwd=2
: 線分の幅．希望する線の幅，標準の線幅の倍数となる．lines() などを伴って描かれる線と同様に軸線にも影響する．

col=2
: 点・線・テキスト・塗りつぶし領域，及び画像に使われる色．これらのグラフ要素は指定可能な色のリストを持ち，このパラメータの値はそのリストへの添字である．当然ながらこのパラメータは限られた範囲のデバイスのみに適用できる．

`font=2` テキストに対して使われるフォントを指定する．可能ならばデバイスドライバが，1 には通常のテキスト，2 には太字，3 にはイタリック体が，そして 4 には太字イタリック体が対応するように調整する．

`font.axis`
`font.lab`
`font.main`
`font.sub` それぞれ，軸の注釈，x, y-軸のラベル，表題・副題に使われるフォント．

`adj=-0.1` 作図位置に関するテキストの位置調整．作図位置に対して，0 は左詰め，1 は右詰め，0.5 は水平中央．実際の値は作図位置の左に現れるテキストに比例し，そのため値 `-0.1` はテキストと作図位置の間に，テキスト幅の 10% の隙間を作る．

`cex=1.5` 文字の拡大率．値は既定の文字の大きさに対する，希望するテキスト文字 (作図文字を含む) のサイズ．

12.5.2 軸と目盛マーク

多くの R の高水準作図は軸を持っており，低水準の `axis()` 作図関数で好みの軸を構成することができる．軸は 3 つの主要な要素を持つ :「軸線」(線種は `lty` 作図パラメータで制御される)，「刻みマーク」(軸線を単位長さ毎に刻むマーク)，そして「目盛り」(長さの単位のラベル) である．それらの要素は以下の作図パラメータで調整する事が出来る．

`lab=c(5, 7, 12)`
最初の 2 つの数字は，それぞれ x 軸と y 軸の希望する目盛り間隔．第 3 の値は希望する軸ラベルの文字数単位での長さ (小数点を含む)．このパラメータに小さすぎる値を選ぶと，全ての軸目盛りが丸められて同じ数になる！

`las=1` 軸ラベルの向き．0 は常に軸に対して平行であること，1 は常に水平であること，そして 2 は常に軸に対して垂直であることを意味する．

`mgp=c(3, 1, 0)`
軸要素の位置．第 1 要素はテキスト行数を単位とした，軸ラベルから軸位置までの距離．第 2 要素は刻みラベルまでの距離で，最後の要素は軸位置から軸線までの距離 (通常は 0)．正の数値は作図領域の外側へと測り，負の数値は内側へと測る．

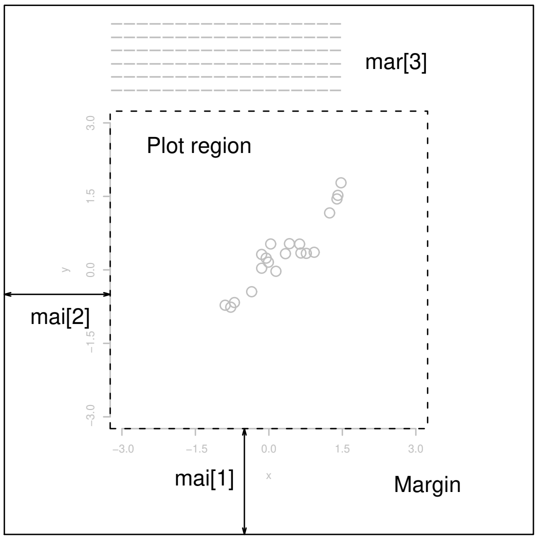
`tck=0.01` 図形領域のサイズの分数としての刻みマークの長さ．`tck` が小さいとき (0.5 より小) `x` 軸と `y` 軸の刻みは同じ大きさにされる．1 の値は格子を与える．負の値は刻みマークを図形領域の外に与える．刻みマークを内部に置くためには `tck=0.01` や `mgp=c(1,-1.5,0)` を使おう．

`xaxs="s"`
`yaxs="d"` それぞれ x 軸と y 軸の軸形式．形式 `\"s"` (standard) と `"e"` (extended) では最小と最大の刻みマークはデータの範囲の外にある．拡張された軸は，どれかの点が端に非常に近いとき，わずかに広げられるかも知れない．この軸の形式は場合によっては，端の近くに大きな空白の隙間を残す．形式 `"i"` (internal) と `"r"` (既定) では刻みマークはデータの範囲に入るが，形式 `"r"` は端付近に若干の空白を残す．

このパラメータを `"d"` (direct axis) にすると，現在の軸を「ロック (*locks in*)」し，その後の全ての (もしくは，少なくともそのパラメータが，上の他の値のどれかに設定されるまで) プロットに対してそれを利用する．固定された尺度の一連のプロットを生成する場合に便利である．

12.5.3 図の余白

R の単一のプロットは「図 (`figure`)」として知られ，余白 (軸ラベル，タイトルなどを含むかも知れない) に囲まれる「作図領域 (*plot region*)」からなり，(普通) 軸そのもので仕切られている．次は典型的な図である．



図表のレイアウトを制御する作図パラメータは以下のものを含む：

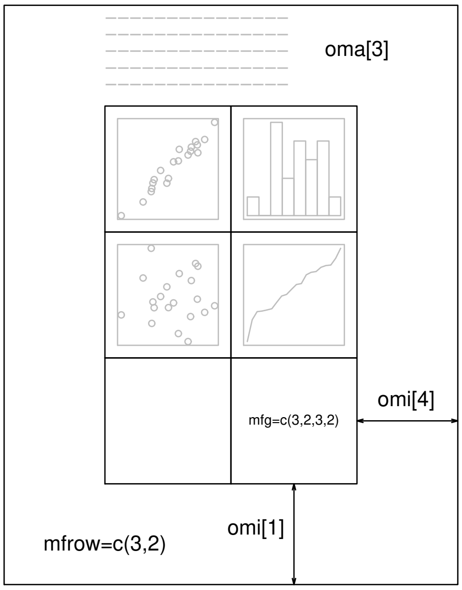
`mai=c(1, 0.5, 0.5, 0)`
: それぞれ下・左・上・右余白の幅で，インチ単位．

`mar=c(4, 2, 2, 1)`
: 単位がテキスト行の高さであることを除き `mai` と同様．

`mar` と `mai` は，どちらかを設定すると他方の値を変化させると言う意味で同等である．このパラメータに選ばれている既定値は通常大きすぎる，つまり右側の余白は滅多に必要でなく，表題が無い場合には上側余白も同じく必要ないためである．下側余白と左側余白は，軸と目盛りラベルを入れるのに十分な大きさである必要がある．更に既定値はデバイス面の大きさを考慮せずに選ばれている．例えば `postscript()` デバイスを引数 `height=4` で利用すると `mar` や `mai` がはっきり指定されない限り，50% の余白を持つプロットになる．複数図表が使われたとき (後をみよ) は，余白は半分に縮小されるが，これは多くの図表が同一ページを共有するには十分ではないかも知れない．

12.5.4 複数の図用の環境

R は図の $n \times m$ 配列を 1 つのページに作成することが可能である．それぞれの図表はそれ自身の余白を持ち，図の配列は，以下の図に示されるように，オプションで「外側余白 (*outer margin*)」で囲まれる．



複数図表に関係する作図パラメータは以下の通りである：

`mfcol=c(3, 2)`
`mfrow=c(2, 4)`
: 複数図の配列の大きさを指定する．第 1 の値は行数で，第 2 は列数である．これらの 2 つのパラメータの唯一の違いは `mfcol` の指定が列順で図を埋め `mfrow` は行順で埋めるということだけである．

  図における配置は `mfrow=c(3,2)` と指定することにより作られるであろう；図は 4 つの図が描かれた後のページを示す

`mfg=c(2, 2, 3, 2)`
: 複数図表環境における現在の図表の位置．最初の 2 つの数値は，現在の図表の行と列，最後の 2 つは複数図表配列にある行と列の数である．配列中の図の間をジャンプするためにこのパラメータを用いよ．同じページの異なった大きさの図表にたいし，「真の」値とは異なる値を後の 2 つの値に使うことすらできる．

`fig=c(4, 9, 1, 4)/10`
: そのページにおける現在の図表の位置．値は左下隅から測ったページの百分率としての左・右・下・上端の位置である．例の値は，ページの右下に図を置くためのものである．ページ内の任意の位置に図表を置くために，このパラメータを指定せよ．

`oma=c(2, 0, 3, 0)`
`omi=c(0, 0, 0.8, 0)`
: 外側余白のサイズ．`mar` や `mai` と同様に，最初はテキスト行単位で数え，2 つ目はインチ単位で下側余白から時計回りに指定する．

外側余白は特にページ毎の表題などのためにとくに有用である．テキストは `outer=T` 引数を付けた `mtext()` 関数で，外側余白に追加することができる．しかしながら，既定では外側余白はなく，従って `oma` や `omi` を使って明示的に作成しなければならない．

より複雑な複数の図の配置は `split.screen()` 関数と `layout()` 関数を使って作り出すことができる．

12.6 デバイスドライバ

R はほとんどいかなるタイプの表示・印刷デバイスにおいても，(異なる品位水準の) グラフを作ることができる．しかしながらその前に R はどんなタイプのデバイスを扱っているのか知らされる必要がある．これは「デバイスドライバ (*device driver*)」を起動することによりなされる．デバイスドライバの目的は R からの作図指示 (例えば「線を引く」) を特定のデバイスが理解できる形式に変換することである．

デバイスドライバはデバイスドライバ関数の呼び出しにより起動される．全てのデバイス指定に対してそのような関数が 1 つある：それらの全てのリストを得るには `help(Devices)` と入力せよ．例えば，次の命令を実行すると

```
> postscript()
```

以降の全てのグラフ出力は「ポストスクリプト」形式でプリンターに送られる．一般に使われるデバイスドライバとしては：

`X11()` X11 ウィンドウシステム用．

`postscript()`
「ポストスクリプト」プリンタへの印刷，もしくは「ポストスクリプト」形式のグラフィックファイルの作成用．

`pictex()` LaTeX ファイルを生成する．

あるデバイスの利用が終了した場合には，次の命令を実行して確実にデバイスドライバを停止させる．

```
> dev.off()
```

これにより，そのデバイスがきれいに終了する．例えばハードコピー (hardcopy) デバイスの場合，この命令により全てのページが完結させられ，プリンタに送られる．

12.6.1 文章の製版用の **Postscript** 図式

`postscript()` デバイスドライバ関数の引数にファイルを指定することにより PostScript 形式でグラフを保存できる．

そのプロットは `horizontal=FALSE` 引数を与えない限り横置きであり，`width` や `height` 引数でグラフのサイズを調整できる．(プロットはこれらの大きさに合うように適当にスケールが調整される．) 例えば，命令

```
> postscript("file.ps", horizontal=FALSE, height=5, pointsize=10)
```

はおそらく文書に含めるであろう 5 インチの高さの図表の「ポストスクリプト」コードを含むファイルを生成する．もし命令中のファイルが既に存在すれば上書きされてしまうことを注意することが重要である．このことは，例えそのファイルが同じ R セッションで先に作り出されたものであっても同様である．

ポストスクリプトの多くの利用法は図を別の文章に組み込むことであろう．これは *encapsulated* ポストスクリプトが作り出されるとき最もうまく働く．R は常に標準的な出力を作り出すが，しかしながら `onefile=FALSE` 引数が与えられたときだけ出力をそのようなものとしてマークする．この普通でない表記は S との互換性のために使われている．その真の意味は出力を単一のページにする (EPSF の仕様の一部) ことを意味する．従って取り込み用のプロットを作るには例えば次のようにする．

```
> postscript("plot1.eps", horizontal=FALSE, onefile=FALSE,
        height=8, width=6, pointsize=10)
```

12.6.2 複数のグラフィックスドライバ

R の高度な利用においては，同時に使用される複数の作図デバイスを持つことがしばしば有益である．もちろん一時にはただ 1 つの作図デバイスだけが作図命令を受け入れることができ，これは「現在のデバイス (current device)」と呼ばれている．複数のデバイスが開かれているとき，それらは任意位置のデバイスの種類を与える名前とともに，番号順の列を作っている．

複数のデバイスを操作するための主要な命令とその意味は以下の通りである：

`X11()` [Unix]

`windows()`
: [Windows]

`Macintosh()`
: [MacOS 8/9]

`postscript()`
`pictex()`
`...`
: デバイスドライバ関数の新規呼出しは新しい作図デバイスを開き，したがってデバイスリストを 1 つ伸長する．このデバイスが現在のデバイスとなり，そこにグラフ出力が送られる．(もっと多くの利用可能なデバイスを持つプラットフォームもあるかもしれない．)

`dev.list()`
: 全てのアクティブなデバイスの番号と名前を返す．リストにおける第 1 位置のデバイスは，常に「無効デバイス (*null device*)」であり，これは全く作図命令を受け付けない．

`dev.next()`
`dev.prev()`
: それぞれ，現在のデバイスの次，および前の位置の作図デバイスの番号と名前を返す．

`dev.set(which=k)`
: 現在のデバイスをデバイスリストの `k` 番目の位置のデバイスに変更するために使うことができる．そのデバイスの番号とラベルを返す．

`dev.off(k)`
: デバイスリストの `k` 番目の位置の作図デバイスを終了させる．`postscript` デバイスなどのいくつかのデバイスに対して，そのデバイスの初期化の方法に依存して，すぐにファイルを印刷するか，後で印刷するために正しくファイルを完結させる．

`dev.copy(device, ..., which=k)`
`dev.print(device, ..., which=k)`
: デバイス *k* のコピーを作る．ここで `device` は `postscript` などのデバイス関数で，

必要ならば ‘...’ で指示される余分な引数を伴う．dev.print は似ているが，複製されたデバイスは直ちに閉じ，ハードコピー印刷等の終了処理が直ちに行われる．

graphics.off()
: 無効デバイスを除き，リスト中の全ての作図デバイスを終了する．

12.7 動的なグラフィックス

R は，点群を回転したり，点を「刷毛塗り」(対話的に強調する) といった，動的なグラフィックスのための組み込み関数を (現時点では) 持っていない．しかしながら，完備した動的なグラフィックス機能が Swayne, Cook そして Buja

による XGobi システム中にある．これは

http://www.research.att.com/areas/stat/xgobi/

から入手でき，R からはパッケージ **xgobi** により使用可能になる．

XGobi は現在 Unix もしくは Windows の X Windows システム上で稼働し，双方に対する R のインターフェイスがある．

Appendix A 入門セッション

以下のセッションは R 環境のいくつかの特徴を，それを使ってみることにより紹介することを目標にしている．システムの多くの特徴は，最初なじみがなく困惑させられるであろうが，すぐに理解できるであろう．このセッションは Unix ユーザ向けに書かれている．

Windows や MacOS Classic の利用者は議論を適当に変更する必要がある．

ログイン，ウィンドウシステムを立ち上げる．データファイル 'morley.data' が作業ディレクトリになければならない．もししなければ，近くの詳しい人に尋ねよ (もしくは既定の R ディレクトリのサブディレクトリ 'base/data' から自分自身で得る)．もしあれば，先に進む．

`$ R` R を使用プラットフォームに即して開始する．

R プログラムが始まり，起動画面が現れる．

(R 内では左側のプロンプトは混乱を避けるために表示しない．)

`help.start()`

オンラインヘルプへの HTML インターフェイスを立ち上げる (使用計算機で利用可能なウエッブブラウザーを使う)．マウスを使ってこの機能の特徴を手短に調べてみよ．

ヘルプウィンドウをアイコン化し，次に進む．

```
x <- rnorm(50)
y <- rnorm(50)
```

x, y 座標を表す 2 組の正規乱数を生成する．

`plot(x, y)`

その点を平面にプロットする．グラフィックスウィンドウが自動的に画面に現れるはずである．

`ls()` R の作業スペースにどういう R オブジェクトがあるか見よ．

`rm(x, y)` 不要なオブジェクトを取り去る (掃除)．

`x <- 1:20` $x = (1, 2, \ldots, 20)$ とする．

`w <- 1 + sqrt(x)/2`

標準偏差の「重み」ベクトル．

```
dummy <- data.frame(x=x, y= x + rnorm(x)*w)
dummy
```

2 つの列ベクトル x と y の「データフレーム」を作りそれを見る．

```
fm <- lm(y ~ x, data=dummy)
summary(fm)
```

y の x への単純線形回帰当てはめを行い，解析結果を眺める．

```
fm1 <- lm(y ~ x, data=dummy, weight=1/w^2)
summary(fm1)
```

標準偏差が分かっているから，重み付き回帰を行うことができる．

`attach(dummy)`

データフレーム中の列ベクトルを変数として見えるようにする．

`lrf <- lowess(x, y)`

ノンパラメトリックな局所回帰関数を作る．

`plot(x, y)`
標準的な点プロット．

`lines(x, lrf$y)`
局所回帰に加える．

`abline(0, 1, lty=3)`
真の回帰直線：(切片 0，傾き 1)．

`abline(coef(fm))`
重み無しの回帰直線．

`abline(coef(fm1), col = "red")`
重み付きの回帰直線．

`detach()` データフレームを検索リストから除く．

```
plot(fitted(fm), resid(fm),
     xlab="Fitted values",
     ylab="Residuals",
     main="Residuals vs Fitted")
```
標準偏差の不均一性をチェックするための標準的な回帰診断プロット．不均一性が認められるだろうか？

`qqnorm(resid(fm), main="Residuals Rankit Plot")`
歪み，尖り，そして外れ値をチェックするための正規スコアプロット．(ここではあまり役にたたない．)

`rm(fm, fm1, lrf, x, dummy)`
再び掃除．

次のセクションは Michaelson と Morley による古典的な光速の測定実験からのデータを調べる．

`file.show("morley.tab")`
オプション．ファイルを眺める．

```
mm <- read.table("morley.tab")
mm
```
Michaelson と Morley データをデータフレームとして読み込み眺める．5 つの実験 (欄 `Expt`) があり，各々は 20 回の試行からなる (欄 `Run`)，そして `sl` は適当に変換された光速値である．

```
mm$Expt <- factor(mm$Expt)
mm$Run <- factor(mm$Run)
```
`Expt` と `Run` を因子に変換する．

`attach(mm)`
データフレームを位置 2(既定) で見えるようにする．

`plot(Expt, Speed, main="Speed of Light Data", xlab="Experiment No.")`
五組の実験を単純な箱型図で比較する．

```
fm <- aov(Speed ~ Run + Expt, data=mm)
summary(fm)
```
「runs」と「experiments」を因子とし，乱塊 (randomized block) として解析する．

```
fm0 <- update(fm, . ~ . - Run)
anova(fm0, fm)
```

「runs」を除いたサブモデルに当てはめを行い，形式的な分散分析を用いて比較する．

```
detach()
rm(fm, fm0)
```

次に移る前に掃除する．

ここで等高線，イメージプロットという作図機能を眺める．

```
x <- seq(-pi, pi, len=50)
y <- x
```

x は区間 $-\pi \leq x \leq \pi$ を 50 個の等間隔に分けるベクトル．y も同様．

```
f <- outer(x, y, function(x, y) cos(y)/(1 + x^2))
```

f は関数 $\cos(y)/(1+x^2)$ の値からなる正方行列で，行と列はそれぞれ x と y に対応する．

```
oldpar <- par(no.readonly = TRUE)
par(pty="s")
```

作図パラメータを保存し，作図領域を「正方形」にする．

```
contour(x, y, f)
contour(x, y, f, nlevels=15, add=TRUE)
```

f の等高線図を描く；詳しく見るために線の数を増やす．

```
fa <- (f-t(f))/2
```

`fa` は f の「非対称部分」(`t()` は転置)．

```
contour(x, y, fa, nlevels=15)
```

等高線図を作る，. . .

```
par(oldpar)
```

. . . そして元の作図パラメータを復帰する．

```
image(x, y, f)
image(x, y, fa)
```

高密度のきれいなイメージプロットを作る (もし希望すればハードコピーを取ることができる)，. . . .

```
objects(); rm(x, y, f, fa)
```

. . . そして次に移る前に掃除する．

R は複素数の演算もできる．

```
th <- seq(-pi, pi, len=100)
z <- exp(1i*th)
```

`1i` は虚数 i の代わりに使われる．

```
par(pty="s")
plot(z, type="l")
```

複素数引数のプロットとは，虚数部を実数部にたいしてプロットすることを意味する．結果は円になるはずである．

`w <- rnorm(100) + rnorm(100)*1i`
: 単位円の内部から標本点を取りたいとする．1 つの方法は標準正規分布に従う実数部と虚数部を持つ複素数を取り . . .

`w <- ifelse(Mod(w) > 1, 1/w, w)`
: . . . そして，円の外部にあるものは逆数を取って円の内部に写像することである．

```
plot(w, xlim=c(-1,1), ylim=c(-1,1), pch="+",xlab="x", ylab="y")
lines(z)
```

すべての点は円の内部にあるが，その分布は一様ではない．

```
w <- sqrt(runif(100))*exp(2*pi*runif(100)*1i)
plot(w, xlim=c(-1,1), ylim=c(-1,1), pch="+", xlab="x", ylab="y")
lines(z)
```

2 つ目の方法は一様分布を使う．点は今やより円の内部に均一に散らばるように見える．

`rm(th, w, z)`
: 再び掃除．

`q()`
: R プログラムを終了する．R の作業スペースを保存するかどうか尋ねられる．こうした探索的なセッションでは，保存する必要はおそらく無いであろう．

Appendix B R を起動する

B.1 UNIX で R を起動する

命令 ‘R’ は R の主プログラムを次の形式により様々なオプション付きで開始したり，

R [*options*] [<*infile*] [>*outfile*],

もしくは，R CMD インターフェイス経由で，直接呼び出されることを意図していない様々な R の道具 (例えば R のドキュメント形式で書かれたファイルの処理や，追加パッケージの操作) への橋渡しとして使うことができる．

ほとんどのオプションは R セッションの最初に何が起こるかを制御する．起動のメカニズムは以下の通りである (オンラインヘルプのトピック ‘Startup’ を参照)．

- ‘--no-environ’ が与えられないと，R は環境変数を設定するためにユーザファイルとサイトファイルを探す．サイトファイルの名前は環境変数 R_ENVIRON で指定する．もしその環境変数がなければ，‘$R_HOME/etc/Renviron.site’ が (もし存在すれば) 使われる．

  ユーザファイルは ‘.Renviron’ で，カレントディレクトリ，次にユーザのホームディレクトリの中で探される．これらのファイルは，‘*name=value*’ の形式の行を含んでいなければならない．(正確なことは help(Startup) を見よ．) 設定したくなるような変数としては，R_PAPERSIZE (既定用紙サイズ)，R_PRINTCMD (既定印刷命令)，R_LIBS(追加パッケージを探す R のライブラリツリーのリスト) がある．
- オプション ‘--no-site-file’ が与えられない限り，R はサイト全体の起動プロファイルを探す．このファイルの名前は環境変数 R_PROFILE から決められる．

  環境変数がない場合は，既定の ‘$R_HOME/etc/Rprofile.site’ がもし存在すれば使用される．
- それから，オプション ‘--no-init-file’ が与えられない限り，R は ‘.Rprofile’ という名前のファイルを現ディレクトリもしくはユーザのホームディレクトリを (この順で) 探し，それを読み込む．
- 同様に，もしあれば，保存されたイメージファイル ‘.RData’ を

  (オプション ‘--no-restore’ または ‘--no-restore-data’ が指定されない限り) 読み込む．
- 最後に，もし関数 .First が存在すれば，それが実行される．この関数 (R セッションの最後に実行される .Last と同様に) は適当な初期化ファイル中に定義されるか，もしくは ‘.RData’ に置いておくことができる．.

加えて，R プロセスが利用可能なメモリを制御するオプションがある (詳細はオンラインヘルプの項目 ‘Memory’ を見よ．)

ユーザは，R によって使用されるメモリ量を制限しようとする場合を除いて，普通はこれらを使う必要はない．

R は以下の命令行オプションを受け入れる．

‘--help’
‘-h’ 　短いヘルプメッセージを標準出力に出力し，正常終了する．

‘--version’
　バージョンメッセージを標準出力に出力し，正常終了する．

'RHOME' R の「ホームディレクトリ」へのパスを標準出力に出力し，正常終了する．シェルスクリプトと man ページを除いて，R のインストールは全て (実行可能プログラム，パッケージ等) をこのディレクトリに置く．

'--save'
'--no-save'
R セッションの終りにデータセットを保存するかどうかを制御する．対話的セッションでどちらも与えられないと，*q()* で終了する際どうするかを尋ねられる．バッチモードではどちらかを指定しなければならない．

'--no-environ'
環境変数を設定するための如何なるユーザファイルも読み込まない．

'--no-site-file'
サイト全体の起動プロファイルの読み込みを抑制する．

'--no-init-file'
ユーザのプロファイルファイルの読み込みを抑制する．

'--restore'
'--no-restore'
'--no-restore-data'
保存されたイメージ (R を開始したディレクトリのファイル '.Rdata') を回復するかどうかを制御する．既定では読み込む．('--no-restore' は個々の '--no-restore-*' オプションを合わせた意味を持つ．)

'--no-restore-history'
履歴ファイル (ふつうは R を開始したディレクトリにあるファイル '.Rhistory' だが，環境変数 R_HISTORY によって指定できる．) を起動時に回復するかどうかを制御する．既定では読み込む．

'--vanilla'
オプション '--no-save'，'--no-environ'，'--no-site-file'，'--no-init-file' そして '--no-restore' を併せたもの．

'--no-readline'
readline を用いた行命令編集をしない．これは Emacs から ESS ("Emacs Speaks Statistics") パッケージを使って R を動かすとき有用である．詳細は Appendix C [行エディタ], page 88 を参照．

'--min-vsize=*N*'
'--max-vsize=*N*'
可変長オブジェクト用のメモリ量の下限や上限を，"vector heap" サイズを *N* バイトに設定することにより指定する．ここで *N* は整数か，最後が 'G', 'M', 'K' もしくは 'k' で終る ('Giga' (2^30), 'Mega' (2^20), (computer) 'Kilo' (2^10), もしくは regular 'kilo'(1000) を意味する) 整数でなければならない．

'--min-nsize=*N*'
'--max-nsize=*N*'
固定長オブジェクト用のメモリ量を，*N* に "cons cell" の数を設定することにより指定する．*N* については上のオプションを参照せよ．ひとつの cons cell は 32-bit マシンでは 28 バイトを，64-bit マシンでは普通 56 バイトを占める．

'--quiet'
'--silent'
'-q' 起動時メッセージを出力しない．

'--slave' R をできるだけ静かに動かす．このオプションは R を補助的に使用するプログラムを支援することを意図している．

'--verbose'
途中結果をもっと多く出力する．特に，R の verbose オプションを TRUE に設定する．R コードはこのオプションを診断メッセージの表示を制御するために用いる．

'--debugger=*name*'
'-d *name*' デバッガー *name* を通して R を動かす．この場合，それ以上の命令行オプションは無視され，代わりにデバッガーの内部から R を実行する際に与える必要があることを注意せよ．

'--gui=*type*'
'-g *type*'
type をグラフィカルユーザインターフェイスとして使う．現在のところ，*type* として使えるのは 'X11'(既定), GNOME サポートが使用可能なら 'gnome'，そして 'none' である．

入出力は通常のようにリダイレクト可能なこと ('<' と '>' を用い) を注意せよ．

命令 R CMD は R と一緒に使うと有用な様々な道具の起動を可能にするが，「直接」使うことを企図してはいない．その一般形は次のようである．

```
R CMD command args
```

ここで *command* は道具の名前で，*args* はそれに渡される引数である．

現在次のような道具が利用できる．

BATCH R をバッチモードで動かす．

COMPILE R と一緒に使うためにコンパイルする．

SHLIB 動的読み込みのために共有ライブラリを作る．

INSTALL 追加パッケージをインストールする．

REMOVE 追加パッケージを取り除く．

build 追加パッケージを作る．

check 追加パッケージを検査する．

LINK 実行可能なプログラムを作成するためのフロントエンド．

Rprof R のプロファイリング用ファイルを後処理する．

Rdconv Rd 書式を様々な他の書式 (HTML, Nroff, LaTeX, plain text, そして S 文章書式を含む) に変換する．

Rd2dvi Rd 書式を DVI/PDF に変換する．

Rd2txt
Rd 書式をテキストに変換する．

Sd2Rd S の文章を Rd 書式に変換する．

最初の 5 つの道具 (つまり BATCH, COMPILE, SHLIB, INSTALL そして REMOVE) はまた CMD オプション無しで R *command args* の形式で「直接」起動できる．

R CMD インターフェイス経由で利用できる各道具の使用法の情報は

```
R CMD command --help
```

で得られる．

B.2 Windows で R を起動する

Windows における起動手順は UNIX におけるそれと非常に近いが，全く同じと言うわけではない．R の Windows 版には 2 種類ある．コンソール版 (RGui.exe) と，主としてバッチ処理用に使われることを意図した端末版 (Rterm.exe) である．

多くのオプションは R セッションの始めと終りに何が起こるかを制御する．起動の機構は次のようである (トピック 'Startup' に関するオンラインヘルプも参照せよ)．

Windows では 'ホームディレクトリ' が常に定義されているとは限らないので，それをどう参照するかはっきりさせておく．もし環境変数 R_USER が定義されていればそれがホームディレクトリになる．次に，環境変数 HOME が定義されていればこれがホームディレクトリになる．さもなくて，環境変数 HOMEDRIVE と HOMEPATH が定義 (WindowsNT/2000 では通常定義されている) されていればこれがホームディレクトリを定義する．これら全てが該当しなければ，起動ディレクトリがホームディレクトリとされる．

- '--no-environ' が与えられないと，R は環境変数を設定するためにユーザファイルとサイトファイルを探す．サイトファイルの名前は環境変数 R_ENVIRON で指定する．もしその環境変数がないか空ならば '$R_HOME/etc/Renviron.site' が (もし存在すれば) 使われる．ユーザファイルは '.Renviron' で，カレントディレクトリ，次にユーザのホームディレクトリの中で探される．これらのファイルは，'*name=value*' の形式の行を含んでいなければならない．(正確なことは help(Startup) を見よ．)

  環境変数はまた対 '*name=value*' の形で命令行の最後においても良い．
- R は行命令オプション '--no-site-file' が与えられない限りサイト全体用の起動プロファイルを探す．このファイルの名前は環境変数 R_PROFILE の値から決まる．もしこの変数が設定されていないと，

  既定の '$R_HOME/etc/Rprofile.site' が，もし存在すれば使用される．
- 次にオプション '--no-init-file' が与えられない限り，R は '.Rprofile' という名前のファイルを現在のディレクトリかユーザのホームディレクトリをこの順で探し，それを読み込み設定を行う．
-

  またもしそれがあれば，保存されたイメージをファイル '.RData' から読み込む (オプション '--no-restore' か '--no-resotre-data' が指定されない限り)．
- 最後にもし関数 .First が存在すればそれが実行される．この関数は (R セッションの最後に実行される関数 .Last と同様に) 適当な起動プロファイル中に定義しておくか，'.RData' においても良い．

加えて R プロセスが利用するメモリを制御するオプションがある (詳細はトピック 'Memory' に対するオンラインヘルプを参照せよ)．

ユーザは普通これらを使用する必要はない．Windows 用の R には後で述べるように既定で R のプロセスが使用するメモリを制限する

'--max-mem-size' がある．

Windows 用の R は次の行命令オプションを持つ．

'--version'
: バージョン情報を出力し終了する．

'--mdi'
'--sdi'
'--no-mdi'
: Rgui が MDI プログラムとして動作

(既定動作はひとつの主ウィンドウの内部で複数の子ウィンドを使う)，また SDI アプリケーション (コンソール，グラフィックス，文章用に複数のトップレベルのウィンドウを使う) として動作するかどうかを制御する．

'--save'
'--no-save'
: R セッションの終りにデータセットを保存するかどうかを制御する．対話的セッションでどちらも与えられないと，*q()* で終了する際，どうするかを尋ねられる．バッチモードではどちらかを指定しなければならない．

'--no-site-file'
: サイト全体の起動プロファイルの読み込みを抑制する．

'--no-init-file'
: 現ディレクトリもしくはユーザの '.Profile' ファイルの読み込みを抑制する．

'--no-environ'
: ファイル '.Renviron' の読み込みを抑制する．.

'--no-restore'
'--no-restore-data'
: 保存されたイメージ (R を開始したディレクトリのファイル '.Rdata') を回復するかどうかを制御する．既定では読み込む．('--no-restore' は個々の '--no-restore-*' オプションを合わせた意味を持つ．)

'--no-restore-history'
: 履歴ファイル (ふつうは R を開始したディレクトリにあるファイル '.Rhistory' だが，環境変数 R_HISTORY によって指定できる．) を起動時に回復するかどうかを制御する．既定では読み込む．

'--vanilla'
: オプション '--no-save', '--no-restore', '--no-site-file', '--no-init-file' そして '--no-environ' を併せたもの．

'--min-vsize=*N*'
'--max-vsize=*N*'
: 可変長オブジェクト用のメモリ量の下限や上限を，"vector heap" サイズを N バイトに設定することにより指定する．ここで N は整数か，最後が 'G', 'M', 'K' もしくは 'k' で終る ('Giga' (2^30), 'Mega' (2^20), (computer) 'Kilo' (2^10), もしくは regular 'kilo'(1000) を意味する) 整数でなければならない．

'--min-nsize=*N*'
'--max-nsize=*N*'
固定長オブジェクト用のメモリ量を，*N* に "cons セル" の数を設定することにより指定する．*N* については上のオプションを参照せよ．ひとつの cons セルは 28 バイトを占める．

'--max-mem-size=*N*'
'--quiet'
'--silent'
'-q' 起動時メッセージを出力しない．

'--slave' R をできるだけ静かに動かす．

'--verbose'
途中結果をもっと多く出力する．特に，R の `verbose` オプションを `TRUE` に設定する．R コードはこのオプションを診断メッセージの表示を制御するために用いる．

'--ess'

ESS の `R-inferior-mode` で使うため `Rterm` をセットアップする．

命令 `Rcmd` は R と一緒に使うと有用な様々な道具の起動を可能にするが，「直接」使うことを企図してはいない．その一般形は次のようである．

```
Rcmd command args
```

ここで *command* は道具の名前で，*args* はそれに渡される引数である．

現在次のような道具が利用できる．

`INSTALL` add-on パッケージをインストールする．

`REMOVE` add-on パッケージを取り除く．

`SHLIB`
`dyn.load`と共に使うために DLL を作る．

`BATCH` R をバッチモードで動かす．

`build` add-on パッケージを作る．

`check` add-on パッケージを検査する．

`Rdconv` Rd フォーマットを様々な他のフォーマット (HTML, Nroff, LaTeX, plain text, そして S 文章フォーマットを含む) に変換する．

`Rd2dvi.sh`
Rd フォーマットを DVI/PDF に変換する．

`Rd2txt`
Rd フォーマットをテキストに変換する．

`Sd2Rd` S の文章を Rd フォーマットに変換する．

`Rcmd` インターフェイス経由で利用できる各道具の用法情報は

```
Rcmd command --help
```

で得られる．

B.3 MacOS Classic で R を起動する

この章は classic MacOS に適用される．MacOS X への UNIX 版の移植についてではない．

R アプリケーションの起動は UNIX や Windows と同様である．Macintosh の R は端末版のみであり，また `R CMD` に相当するものはない．

R アプリケーションアイコン上でダブルクリックすると小さいウィンドウが現れる．これは他の移植での命令行に相当する．ここで起動時のオプションを入力することもできるし，単に ⟨RET⟩ を押せば既定のオプションで R が起動する．R のメインプログラムはこの章であとに述べるように色々なオプションで次のような形で起動できる．

`R` [*options*]

Macintosh 標準のインターフェースで UNIX と同じ [<*infile*] [>*outfile*] を選択することもできるが，これらの組込みは現時点で完全ではないので知らなくて良い．

ほとんどのオプションは R セッションの最初に何が起こるかを制御する．起動のメカニズムは以下の通りである (オンラインヘルプのトピック '`Startup`' を参照)．

- '`--no-environ`' が与えられない限り，R は R のアプリケーションフォルダの '`etc`' サブディレクトリでファイル '`.Renviron`' を探す．見付かったファイルは環境変数を設定するのに使われる．変数は '*`name=value`*' の形式の行としておかれる．(正確な記述は `help(Startup)` を参照．)
- それから，オプション '`--no-init-file`' が与えられない限り，R は '`.Rprofile`' という名前のファイルを R のアプリケーションフォルダの '`etc`' サブディレクトリで探し，それを R のコードとして処理する．
- 同様に，もしあれば，保存されたイメージファイル '`.RData`' を (オプション '`--no-restore`' か '`--no-restore-data`' が指定されない限り) ロードする．このファイルは R のアプリケーションフォルダの '`etc`' サブディレクトリで探す．
- 最後に，もし関数 `.First` が存在すれば，それが実行される．この関数 (R セッションの最後に実行される `.Last` と同様に) は適当な初期化ファイル中に定義されるか，もしくは '`.RData`' に置いておくことができる．.

加えて，R プロセスが利用可能なメモリを制御するオプションがある (詳細はオンラインヘルプの項目 '`Memory`' を見よ．) ユーザは，R によって使用されるメモリ量を制限しようとする場合を除いて，普通はこれらを使う必要はない．MacOS の場合の注意として，他の Macintosh のソフトで通常おこなわれるとおり，あらかじめ 'Finder/Information' から R アプリケーションに割り当てられるメモリを予約しておかなければならない．base 配布のデモと例を実行するためには 32000k あれば十分である．

R は以下の命令行オプションを受け入れる．

'`--version`'
: バージョン情報を標準出力に出力し，正常終了する．

'`--save`'
'`--no-save`'
: R セッションの終りにデータセットを保存するかどうかを制御する．対話的セッションでどちらも与えられないと，*`q()`* で終了する際どうするかを尋ねられる．バッチモードではどちらかを指定しなければならない．

'--no-init-file'
: ユーザのプロファイルファイルの読み込みを抑制する．

'--restore'
'--no-restore'
'--no-restore-data'
: 保存されたイメージ（R を開始したディレクトリのファイル'`:etc:.Rdata`'）を回復するかどうかを制御する．既定では読み込む．('--no-restore' は個々の '--no-restore-*' オプションを合わせた意味を持つ．)

'--no-restore-history'
: 履歴ファイル (ふつうは R を開始したディレクトリにあるファイル'`.Rhistory`'だが，環境変数 `R_HISTORY` によって指定できる．) を起動時に回復するかどうかを制御する．既定では読み込む．

'--vanilla'
: オプション '--no-save'，'--no-environ'，'--no-site-file'，'--no-init-file' そして '--no-arestore' を併せたもの．

'--min-vsize=*N*'
'--max-vsize=*N*'
: 可変長オブジェクト用のメモリ量の下限や上限を，"vector heap" サイズを N バイトに設定することにより指定する．ここで N は整数か，最後が 'G', 'M', 'K' もしくは 'k' で終る ('Giga' (2^30), 'Mega' (2^20), (computer) 'Kilo' (2^10), もしくは regular 'kilo'(1000) を意味する) 整数でなければならない．

'--min-nsize=*N*'
'--max-nsize=*N*'
: 固定長オブジェクト用のメモリ量を，"cons cell" の数を N に設定することにより指定する．N については上のオプションを参照せよ．ひとつの cons cell は 32-bit マシンでは 28 バイトを，64-bit マシンでは普通 56 バイトを占める．

'--quiet'
'--silent'
'-q'
: 起動時メッセージを出力しない．

R をできるだけ静かに動かす．

'--verbose'
: 途中結果をもっと多く出力する．特に，R の `verbose` オプションを `TRUE` に設定する．R コードはこのオプションを診断メッセージの表示を制御するために用いる．

Appendix C 行エディタ

C.1 準備

R を UNIX のもとでコンパイルする際 GNU **readline** ライブラリが利用可能なら，再呼び出し，編集そして再実行が可能な組み込みの行編集エディタが利用可能になる．

注意：この補遺は通常の命令行インターフェースに関するもので，UNIX の GNOME インターフェースには適用「されない」．

起動時にオプション '`--no-readline`' を使うとこの機能は無効になる (ESS[1] と一緒に使うとき便利)．

Windows 版の R はもう少し単純な行編集エディタを持つ:GUI の '`Help`' メニューから '`Console`' を見よ．

また `Rterm.exe` での行編集についてはファイル '`README.Rterm`' にある．

R を **readline** 機能とともに使うと以下に説明する関数が使用可能になる．

これらの多くは Control もしくは Meta 文字を使う．*`Control-m`* といった制御文字は ⟨CTRL⟩ キーを押しながら ⟨m⟩ キーを押すことにより得られ，以下では *`C-m`* と書かれる．*`Meta-b`* といったメタ文字は ⟨META⟩ キーを押しながら ⟨b⟩ キーを押すことにより得られ，以下では *`M-b`* と書かれる．もし使用計算機が ⟨META⟩ キーを持たなければ，*`ESC`* キーで始まる 2 文字列を使うことにより Meta 文字を入力できる．したがって，*`M-b`* を入力するには ⟨ESC⟩⟨b⟩ を入力する．*`ESC`* 文字列はまた真の Meta キーを持つ機械でも使用できる．Meta 文字では大小文字の区別が意味を持つことを注意しよう．

C.2 編集動作

R プログラムはタイプされた命令の履歴を保存しており，履歴中の命令は再呼び出しでき，必要なら変更し，新しい命令として再実行できる．Emacs 流の行編集エディタではこの編集動作中の如何なる直接的なタイプは編集中の命令に挿入され，カーソルの右にある文字列を移動する．*vi* モードでは，文字挿入モードは *`M-i`* もしくは *`M-a`* で始まり，文字がタイプされ，挿入モードは ⟨ESC⟩ をタイプすることにより終了する．

⟨RET⟩ を押すと何時でも命令が再実行される．

他の編集動作は以下の表にまとめられている．

不幸にも，例えば矢印キーといった移動キーを使えるようにすることは不可能のように見え，しばしば面倒を引き起こす．

C.3 行エディタの要約

[1] 'Emacs Speaks Statistics' パッケージ；URL `http://ess.stat.wisc.edu/` を見よ．

命令再呼び出しと垂直移動

C-p 次の命令に移る (履歴をさかのぼる).

C-n 次の命令に移る (履歴を下がる).

C-r text 文字列 *text* を含む直近の命令を探す.

ほとんどの端末では *C-p* と *C-n* の代わりに，それぞれ上向き・下向き矢印キーを使うことができる.

カーソルの水平移動

C-a

命令の先頭に移動.

C-e 行末に移動.

M-b 1 語分戻る.

M-f 1 語分進む.

C-b 1 文字分戻る.

C-f 1 文字分進む.

ほとんどの端末では *C-b* と *C-f* の代わりに，それぞれ左向き・右向き矢印キーを使うことができる.

編集と再実行

text カーソル位置に *text* を挿入.

C-f text カーソル位置の後に *text* を挿入.

⟨DEL⟩ 前の文字を削除 (カーソルの左側)

C-d カーソル位置の文字を削除.

M-d カーソル位置の単語を削除し，それを「保存」する.

C-k カーソルから命令末尾までを削除し，それを「保存」する.

C-y 直近に保存されたテキストをここに挿入 (yank).

C-t カーソル位置の文字を次の文字と交換.

M-l 単語の残りの部分を小文字化.

M-c

単語の残りの部分を大文字化.

⟨RET⟩ 命令を再実行.

最後の ⟨RET⟩ は行命令編集列を終了させる.

Appendix D 関数と変数の索引

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Appendix E 概念の索引

1

2

A

B

C

D

E

F

G

I

K

L

M

N

O

P

Q

R

S

V

W

Appendix F 参考文献

D. M. Bates and D. G. Watts (1988), *Nonlinear Regression Analysis and Its Applications.* John Wiley & Sons, New York.

Richard A. Becker, John M. Chambers and Allan R. Wilks (1988), *The New S Language*[1]. Chapman & Hall, New York. この本はしばしば「青本 (*Blue Book*)」と呼ばれる．

John M. Chambers and Trevor J. Hastie eds. (1992), *Statistical Models in S*[2]. Chapman & Hall, New York. この本はしばしば「白本 (*White Book*)」と呼ばれる．

Annette J. Dobson (1990), *An Introduction to Generalized Linear Models*[3]. Chapman and Hall, London.

Peter McCullagh and John A. Nelder (1989), *Generalized Linear Models.* Second edition, Chapman and Hall, London.

John A. Rice (1995), *Mathematical Statistics and Data Analysis.* Second edition. Duxbury Press, Belmont, CA.

S. D. Silvey (1970), *Statistical Inference.* Penguin, London.

[1] 訳注：邦訳『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 I，II』，渋谷政昭・柴田里程訳，共立出版 (1991)

[2] 訳注：邦訳『S と統計モデル ―データ科学の新しい波―』，柴田里程訳，共立出版 (1994)

[3] 訳注：邦訳『統計モデル入門』，田中豊他訳，共立出版 (1993)