第1章　R の準備

本章では R を利用するための必要な知識をまとめます。R 自体はとても奥が深い言語であり、その全体を本章だけでまとめるには無理です。ただし R を道具として位置づけ、クラスタリングに R を使うという目的に限定すれば、ここで説明する事項で十分です。R はベクトルと行列の演算が本質的であり、そこさえ押さえておけば、十分役に立つ道具となりえます。

1.1　R のインストール

R は Windows の他 Linux や Mac OS でも動作します。ソースも公開されているので、コンパイルすることも可能ですが、バイナリでインストールした方が簡単です。以下のサイトからダウンロードできます。

`The R Project ： http://www.r-project.org/`

日本では以下でミラーされているので、直接、以下の場所からダウンロードするのもよいです。

会津大学　`ftp://ftp.u-aizu.ac.jp/pub/lang/R/CRAN`
東京大学　`ftp://ftp.ecc.u-tokyo.ac.jp/CRAN/`
筑波大学　`http://cran.md.tsukuba.ac.jp/`

Windows であれば、`R-2.5.1-win32.exe` をダウンロードします[1]。
`R-2.5.1-win32.exe` をクリックすると、インストーラーが起動されます。デフォルトの設定で特に問題はないので、クリックしてゆけばインストールできます。ただし本書ではインストール先を `c:\R\R-2.5.1` と仮定して記述しています。

1.2　R の基本操作

1.2.1　R の起動と終了

この節の目標は R をとりあえず使ってみて、R の概略を把握することです。R の操作を少しでも知っていたら、この章を読む必要はありません。

[1] 2007 年 7 月 12 日の時点での最新版です。

まず R の起動ですが、ディスクトップ上にアイコンができているでしょうから、それをクリックします。以下のようなウィンドウが開きます。

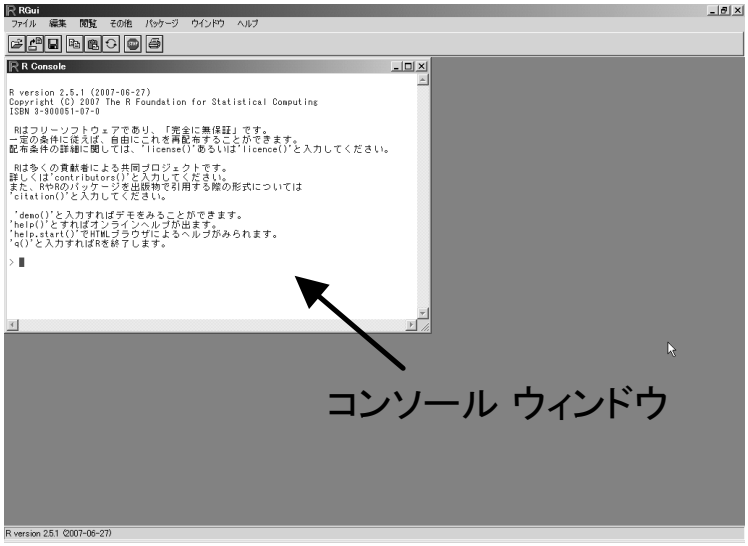

図 1.1: R の起動

図で示した R Console のウィンドウが実際の作業場所です。Windows の「コマンドプロンプト」（DOS 窓）のようなものです。R Console の中の > がプロンプトです。この後に R のコマンドを書いて Enter を入力すれば、そのコマンドが実行されて、結果が表示されます。ただし結果を変数に代入した場合、結果は表示されません。変数の中身を見るには、変数の後に Enter を入力すれば、その変数の中身が表示されます。結果の表示が終われば、またプロンプト> が表示され、次のコマンドが入力できる状態になります。つまり、コマンド、結果、コマンド、結果、··· の繰り返しです。

試してみます。

```
> a <- 2        # 変数 a に 2 を代入、変数の中身は表示されない
> b <- 3        # 変数 b に 3 を代入、変数の中身は表示されない
> a             # 変数 a の中身は >a [RET] で表示される
[1] 2
> b             # 変数 b の中身は >b [RET] で表示される
[1] 3
> a <- 2; b <- 3  # ; で区切って 1 行に複数のコマンドもかける
> (a <- 2)      # 括弧をつけると、表示も行われる
[1] 2
> a + b
```

```
[1] 5          #  a + b の結果が表示される
> a - b
[1] -1         #  a - b の結果が表示される
> a * b
[1] 6          #  a * b の結果が表示される
> a / b
[1] 0.6666667     #  a / b の結果が表示される
> b %% a
[1] 1          #  剰余は %% です
> a ** b
[1] 8          #  累乗は ** です。^ も使えます。
```

演算の優先順位は数学のものと同じです。複雑な場合は括弧を付けた方が無難です。

```
> a + 3 * a - b
[1] 5
> a + 3 * (a - b)
[1] -1
```

また、コマンドの履歴や編集は Emacs ライクなキーが割り当てられています。Ctrl-p と Ctrl-n で以前に入力したコマンドを呼び出せます。Ctrl-b でカーソルを前方に、Ctrl-f でカーソルを後方に移動できます。また Ctrl-a で行頭に、Ctrl-e で行末にカーソルを移動できます。また Ctrl-k で文末まで削除できます。他にもいろいろありますが、この程度、知っていればよいでしょう。

適当に処理が終わったら、R を終了させます。終了させるには、q() という関数を実行するか、メニューバーから [ファイル] → [終了] です。

```
> q()
```

終了時に以下のように作業スペースを保存するかどうか聞かれます。保存した場合は、その保存ファイルを呼び出すことで、今回定義した変数などが再定義なしで使えます。

図 1.2: 作業スペースの保存

保存した場合、次回、R を立ち上げた後で、メニューバーの [ファイル] → [作業スペースの読み込み] から保存した作業スペースを読み込めます。作業スペースはワーキングディレクトリに保存されるので、通常は R をインストールしたディレクトリに保存されています。ファイルの種類が R image になっているので判断できると思います。

```
> load("C:\\R\\R-2.5.1\\.RData")   ## 作業スペースの読み込みの関数
> a         # 先に定義した変数 a と b が再定義なしでつかる。
[1] 2
> b
[1] 3
```

コマンドの履歴は作業スペースを保存したときに、同時に保存されます。現在残っている履歴のファイルが R の起動時に自動的に読み込まれます。

1.2.2　R の関数

三角関数や *log* などの基本的な関数は R で準備されています。

```
> log(12.3)       # log の底は e です。
[1] 2.509599
> log10(12.3)     # 底が 10 の log
[1] 1.089905
> log2(12.3)      # 底が 2 の log
[1] 3.620586
> log(12.3,base=5)  # 一般の底は base= で指定します。
[1] 1.559302
> exp(1.23)       # e^x は exp(x) と書きます。
[1] 3.421230
```

```
> pi              # 円周率は pi
[1] 3.141593
> sin(pi/3)       #  三角関数も使えます。
[1] 0.8660254
> sqrt(12.3)      #  平方根は sqrt
[1] 3.507136
> round(12.3)     #  丸め（四捨五入）
[1] 12
> floor(12.3)     #  切り捨て
[1] 12
> ceiling(12.3)   #  切り上げ
[1] 13
```

1.2.3 ベクトルの演算

R が扱うデータはベクトルが基本です。

```
> (z <- numeric(8))  # 次元数 (8) を与えた 0 ベクトルの作成
[1] 0 0 0 0 0 0 0 0
> (a <- c(1.2, -0.3, 5, 0, 3.1))  # ベクトルを作るには c() で数値を並べます
[1]  1.2 -0.3  5.0  0.0  3.1
> a[2]            # ベクトル a の第２次元の値
[1] -0.3          # 次元は１から数えることに注意
> length(a)       # ベクトルの次元数は length
[1] 5
> sum(a)          # ベクトルの要素の和
[1] 9
> (b <- c(2:6))  # m から n の連続した整数は m:n
[1] 2 3 4 5 6
> (d <- c(-7:-3))  # : にはマイナスも使えます。
[1] -7 -6 -5 -4 -3
> 2 * a             # ベクトルのスカラー倍
[1]  2.4 -0.6 10.0  0.0  6.2
> a + b
[1] 3.2 2.7 9.0 5.0 9.1  # ベクトルの足し算は各要素どうしの和
> a * b
[1]  2.4 -0.9 20.0  0.0 18.6  # ベクトルの掛け算は各要素どうしの積
> sum(a * b)   # 内積はこれで求まる
[1] 40.1
```

```
>  a %*% b      # 行列の掛け算 %*% を使った内積の計算
     [,1]
[1,] 40.1
```

上記の例は、ベクトル同士の掛け算*を除けば、数学上でも同じ操作を意味するので、特に難しいことはありません。

R でのベクトルの操作で気を付けなければならないのは、数値に対する関数にベクトルを与える場合です。

```
> b
[1] 2 3 4 5 6
> sqrt(b)
[1] 1.414214 1.732051 2.000000 2.236068 2.449490
```

上記の例のように、各要素に対してその関数が適用されることになります。これを応用すると、以下のようなことも可能です。

```
> b
[1] 2 3 4 5 6
> b + 1
[1] 3 4 5 6 7    # 各要素に 1 が足される
> z + 2       # ゼロベクトルに数値を足すと、その数値からなるベクトル
[1] 2 2 2 2 2 2 2 2
```

ベクトルの結合は以下のようにして行えます。

```
> a <- c(1:4)
> b <- c(5:10)
> (d <- c(a,b))   # ベクトル a と b を結合
 [1]  1  2  3  4  5  6  7  8  9 10
> (d <- c(a,b,a)) # ベクトルはいくつでも結合可能
 [1]  1  2  3  4  5  6  7  8  9 10  1  2  3  4
```

またベクトル要素のある部分を取り出したり、削ったりするのも容易です。

```
> b[c(2,4)]   # 2番目と4番目の要素を取り出す
[1] 3 5
> b[-c(1,3)]  # 1番目と3番目の要素を除く
[1] 3 5 6
> b[b > 3]
[1] 4 5 6     # 3より大の要素を取り出す。
> b[(b %% 2) == 0]
[1] 2 4 6     # 偶数の要素だけ取り出す
```

最後の 2 つの例ですが、括弧 [ ] の中に条件式を書く場合、[ ] の中が TRUE と FALSE からなるベクトルになり、TRUE の位置の要素だけが取り出されるということです。

```
> b > 3
[1] FALSE FALSE  TRUE  TRUE  TRUE
> (b %% 2) == 0
[1]  TRUE FALSE  TRUE FALSE  TRUE
```

1.2.4 行列の演算

本質的に R ではベクトルと行列が扱えればよいです。その他のデータ構造（文字列、リストなど）で行うことを、無理に R で行う必要はありません[2]。

行列を作成するのは関数 matrix() にベクトルを与えるのが基本です。

```
> (a <- matrix(c(1:6), ncol=3))  # ncol で列数を指定
     [,1] [,2] [,3]              # 列毎にベクトル要素が埋まる
[1,]    1    3    5
[2,]    2    4    6
> (a <- matrix(c(1:6), nrow=2))  # nrow で行数を指定してもよい
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> (a <- matrix(c(1:6), nrow=2, byrow=TRUE)) # 行毎にベクトル要素を埋める
     [,1] [,2] [,3]                        # には byrow=TRUE
[1,]    1    2    3
[2,]    4    5    6
```

R では複数行に渡ってコマンドを記述できるので、以下のように入力するとわかりやすいかもしれません。

```
> a <- matrix(c(
+
+  1, -2,  1, 0,
+ -1,  3, -1, 2,
+  3,  0,  0, 1
+
+ ), nrow=3, byrow=TRUE)
> a
     [,1] [,2] [,3] [,4]
```

[2]データフレームは使えた方がよいですが、必須ではないと思います。

```
[1,]    1   -2    1    0
[2,]   -1    3   -1    2
[3,]    3    0    0    1
```

上記のコマンド中の改行の後にある + はコマンドが続くことを示すシステムの方で表示している記号です。実際に入力することはありません。

行列中のある部分を取り出したり、削ったりするのは容易です。

```
> (a <- matrix(c(1:30), nrow=5, byrow=TRUE))
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    3    4    5    6
[2,]    7    8    9   10   11   12
[3,]   13   14   15   16   17   18
[4,]   19   20   21   22   23   24
[5,]   25   26   27   28   29   30
> a[3,2]    #  3行2列目の要素
[1] 14
> a[2,]     #  2行目の取り出し、結果はベクトル
[1]  7  8  9 10 11 12
> a[,3]     #  3列目の取り出し、結果はベクトル
[1]  3  9 15 21 27
> a[c(2,4),]  #  2行目と4行目からなる行列
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    7    8    9   10   11   12
[2,]   19   20   21   22   23   24
> a[-c(2,4),]  #  2行目と4行目を取り除いた行列
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    3    4    5    6
[2,]   13   14   15   16   17   18
[3,]   25   26   27   28   29   30
> a[-c(2,4),c(1:4)]  #  2行目と4行目を取り除き、1〜4列目からなる行列
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]   13   14   15   16
[3,]   25   26   27   28
```

既存の行列に行や列を追加するのは、効率の面からやらない方がよいです。新たに目的の行列を作ったり、予め大きな行列を用意しておいて、そこに要素を埋め込んでゆく方がよいです。

行列の演算はベクトルの場合と同様です。

```
> (a <- matrix(c(1:6), nrow=2, byrow=TRUE))
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> (b <- matrix(c(1:6), nrow=2))
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> a + b   # 演算は各要素に対して実行される
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    6    9   12
> a * b   # 掛け算も各要素に対して実行される
     [,1] [,2] [,3]
[1,]    1    6   15
[2,]    8   20   36
> a / b   # 割り算も各要素に対して実行される
     [,1]      [,2] [,3]
[1,]    1 0.6666667  0.6
[2,]    2 1.2500000  1.0
> a - 2   # 各要素に対して実行される
     [,1] [,2] [,3]
[1,]   -1    0    1
[2,]    2    3    4
> a / 2   # 割り算も各要素に対して実行される
     [,1] [,2] [,3]
[1,]  0.5  1.0  1.5
[2,]  2.0  2.5  3.0
> t(b)  # t() は転置する関数
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
> (d <- a %*% t(b))  #  %*% は行列の積
     [,1] [,2]
[1,]   22   28
[2,]   49   64
```

行列に関係する関数は非常にたくさんあります。

```
> dim(d)      # 行列の次元
[1] 2 2
> ncol(d)      # 行列の列数
[1] 2
> nrow(d)      # 行列の行数
[1] 2
> solve(d)    # 逆行列
          [,1]       [,2]
[1,]  1.777778 -0.7777778
[2,] -1.361111  0.6111111
> det(d)  # 行列式
[1] 36
> (ed <- eigen(d))  # 固有値と固有ベクトル
$values
[1] 85.5793377  0.4206623

$vectors
           [,1]       [,2]
[1,] -0.4030412 -0.7920648
[2,] -0.9151818  0.6104370
> ed$values    # 固有値を取り出す
[1] 85.5793377  0.4206623
> ed$vectors   # 固有ベクトルを取り出す
           [,1]       [,2]
[1,] -0.4030412 -0.7920648
[2,] -0.9151818  0.6104370
> d %*% ed$vectors[,1]   # 確認してみる
          [,1]
[1,] -34.49200
[2,] -78.32066
> ed$values[1] * ed$vectors[,1]
[1] -34.49200 -78.32066   #  OK
> diag(d)   # 対角要素の取り出し
[1] 22 64
> diag(d) <- 2   # 対角要素への代入
> d
     [,1] [,2]
[1,]    2   28
[2,]   49    2
```

```
> diag(3)   # 3行3列の単位行列の作成
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

QR 分解と特異値分解はよく使われます。

```
> a
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> (aqr <- qr(a))   # QR 分解は qr()
$qr
           [,1]       [,2]      [,3]
[1,] -4.1231056 -5.3357838 -6.548462
[2,]  0.9701425 -0.7276069 -1.455214

$rank
[1] 2

$qraux
[1] 1.2425356 0.7276069 1.4552138

$pivot
[1] 1 2 3

attr(,"class")
[1] "qr"
> (asvd <- svd(a))   # 特異値分解は svd()
$d
[1] 9.5080320 0.7728696

$u
           [,1]       [,2]
[1,] -0.3863177 -0.9223658
[2,] -0.9223658  0.3863177

$v
           [,1]       [,2]
```

```
[1,] -0.4286671  0.8059639
[2,] -0.5663069  0.1123824
[3,] -0.7039467 -0.5811991
```

1.2.5　ファイル入出力

入力については、データが書かれているファイルを読み込めれば十分でしょう。データは本質的には行列なので、行列の書かれたファイルを読み込んで、R 内部で行列を作ることができればよいです。まず行列の各要素をどのように書くかがポイントですが、ベタに空白文字で区切って以下のように書かれていたとします。ファイル名は data.txt として、作業スペースのディレクトリに置かれているとします。

```
 1.0   2.1  -1.1   0
 0     3     2.2  -1
 2.7  -6     1.8   3.1
```

この場合、関数 `scan()` を使うのが簡単です。

```
> (x <- matrix(scan("data.txt"), ncol=4, byrow=TRUE))
Read 12 items
     [,1] [,2] [,3] [,4]
[1,]  1.0  2.1 -1.1  0.0
[2,]  0.0  3.0  2.2 -1.0
[3,]  2.7 -6.0  1.8  3.1
> dim(x)
[1] 3 4   # 3行4列の行列
```

`ncol=4` の指定が必要なことから明らかですが、data.txt は 1 行あたりの要素数は関係ありません。行列の要素の数値を並べておけばよいです。ただしわかりやすいように上記のように作成しておくのがよいと思います。

出力に対しても、R の行列をベタにファイルに書き出せればよいです。出力は関数 `write()` を使うのが簡単です。

```
> write(x, file="data1.txt",ncolumns=4)
>
```

data1.txt の中身は以下のようになります。

```
1 0 2.7 2.1
3 -6 -1.1 2.2
1.8 0 -1 3.1
```

よく見ると、行と列が逆になっているのが分かります。つまり write() は列をベースに書き出します。byrow=TRUE は指定できないので、行列を転置してから出力すればよいです。

```
> write(t(x), file="data2.txt",ncolumns=4)
>
```

data2.txt の中身は以下のようになります。

```
1 2.1 -1.1 0
0 3 2.2 -1
2.7 -6 1.8 3.1
```

ベクトルのファイルへの書き出しは、そのまま write() を使えばよいです。1 行に 1 要素にしたい場合、以下のようになります。

```
> a <- c(1:100)
> write(a, file="data3.txt",ncolumns=1)
```

data3.txt の中身は以下のようになります。

```
1
2
3
...
99
100
```

文字列などをファイルに混在して出力したい場合には、ファイルに出力を追加してゆく形で write() を使えばよいです。

例えば先の data3.txt の 1 行目に

```
# This is an example
```

と書きたいとします。このときは以下のように行います。

```
> a <- c(1:100)
> write("# This is an example ", file="data3.txt")
> write(a, file="data3.txt",ncolumns=1, append=TRUE)
```

data3.txt の中身は以下のようになります。

```
# This is an example
1
2
3
...
99
100
```

もっと細かいフォーマットで出力したければ、関数 cat() と sink() を利用します。

まず cat() は C 言語でいうところの printf のような関数で、文字列や数値を混在させた出力が可能です。

```
> x <- matrix(c(1:20),ncol=5)
> cat("matrix x\n")
matrix x
> 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
> cat("row of x : ",nrow(x),"\n")
row of x :  4
> cat("col of x : ",ncol(x),"\n")
col of x :  5
```

cat() の出力先は標準出力です。関数 sink() は出力を指定したファイルに変更できます。標準出力に戻す場合には、引数を入れずに単に sink() を実行します。

```
> sink("data4.txt")
> x <- matrix(c(1:20),ncol=5)
> cat("matrix x\n")
> x
> cat("row of x : ",nrow(x),"\n")
> cat("col of x : ",ncol(x),"\n")
> sink()
```

data4.txt の中身は以下のようになります。

```
matrix 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
row of x :  4
col of x :  5
```

1.2.6 統計関係の関数

R は統計解析ソフトですから、統計関係の関数がたくさん用意されています。また実際にそれら関数を使えば、ほとんどの行いたい計算が実行できます。利用頻度の高いものを列挙しておきます。

```
> a <- c(5, 1, 2, 4, 2)
> max(a)   # 最大値
[1] 5
> min(a)   # 最小値
[1] 1
> mean(a)  # 平均
[1] 2.8
> var(a)   # 分散
[1] 2.7
```

`var()` は分散ですが不偏標本分散の方です。

$$var(a) = \frac{1}{n-1}\sum_{i=1}^{n}(a_i - mean(a))^2$$

```
> b <- c(7, 2, 1, 6, 4)
> cov(a,b)  # 共分散
[1] 3.75
> cor(a,b)  # 相関係数
[1] 0.8951436
```

分散に不偏標本分散が使われるので、共分散 `cov()` の定義は以下になります。

$$cov(a,b) = \frac{1}{n-1}\sum_{i=1}^{n}(a_i - mean(a))(b_i - mean(b))$$

相関係数の定義は以下です。

$$cor(a,b) = \frac{cov(a,b)}{sqrt(var(a)) * sqrt(var(b))}$$

データが列ベクトルで表現されている行列の場合、分散共分散行列は以下のようにして求まります。

```
> d <- c(6, 0, 1, 1, 3)
> (x <- matrix(c(a,b,d),nrow=5))
     [,1] [,2] [,3]
[1,]    5    7    6
[2,]    1    2    0
[3,]    2    1    1
[4,]    4    6    1
[5,]    2    4    3
> cov(x,x)
     [,1] [,2] [,3]
[1,] 2.70 3.75 2.80
[2,] 3.75 6.50 4.25
[3,] 2.80 4.25 5.70
```

代表的な確率分布に関しては、その確率密度関数、分布関数、分位点関数、乱数発生関数が用意されています。それぞれの確率分布の名前の前に、d、p、q、r がついた名前になっています。

例えば、正規分布は norm という名前がついており、関数 dnorm() は正規分布の確率密度関数、関数 pnorm() は正規分布の分布関数、関数 qnorm() は正規分布の分位点関数、そして関数 rnorm() は正規分布の乱数発生関数となっています。

```
> dnorm(0.5, mean=1.5, sd=2)  # 平均 1.5 分散 2 の正規分布の確率密度関数で
[1] 0.1760327                 # 0.5 の値
> dnorm(0.5)    # mean と sd が省略されたら、標準正規分布
[1] 0.3520653   # 標準正規分布の確率密度関数で 0.5 の値
> pnorm(0.5)    # 標準正規分布の分布関数で 0.5 の値
[1] 0.6914625
> qnorm(0.975)  # P(X > a) = 0.975 となる a
[1] 1.959964
> rnorm(10)     # 標準正規分布に従う乱数 10 個発生
 [1] -0.5018074  1.8644939  0.1306954  1.1643899 -0.7462392  0.8258899
 [7] -0.9731133 -0.3893751 -1.0051341  1.3273352
```

その他の代表的な確率分布に関しては、以下のようなものがあります。

関数名	確率分布
_beta()	ベータ分布
_binom()	２項分布
_cauchy()	コーシー分布
_chisq()	χ^2 分布
_exp()	指数分布
_f()	F 分布
_gamma()	ガンマ分布
_norm()	正規分布
_pois()	ポアソン分布
_t()	t 分布
_unif()	一様分布

表 1.1: 確率分布

1.2.7 パッケージ

R には標準で使える関数の他にも、様々な解析の目的のために作られた関数が多数あります。それら関数はライブラリと呼ばれます。ある目的のためにライブラリを集めたものがパッケージです。例えば `SparseM` というパッケージは、スパース行列を扱うための関数（ライブラリ）を集めたライブラリのセットです。R には有用なパッケージが多数有り、自分が行いたい解析が実際にパッケージにまとめられている場合も多々あります。

ここではパッケージの導入方法を解説します。

例えば、先にあげた `SparseM` を導入してみます。Windows の場合、パッケージの導入は簡単です。メニューバーから [パッケージ] → [パッケージのインストール] を選びます。すると図のようにパッケージのダウンロード元のサイトのリストが表示されます。

図 1.3: ダウンロード元のサイトの選択

ここで近いサイトを選びます。図では `Japan(Tsukuba)` を選んでいます。すると次にパッケージのリストが表示されますので、ここで目的のパッケージ（`SparseM`）を選べばよいです。

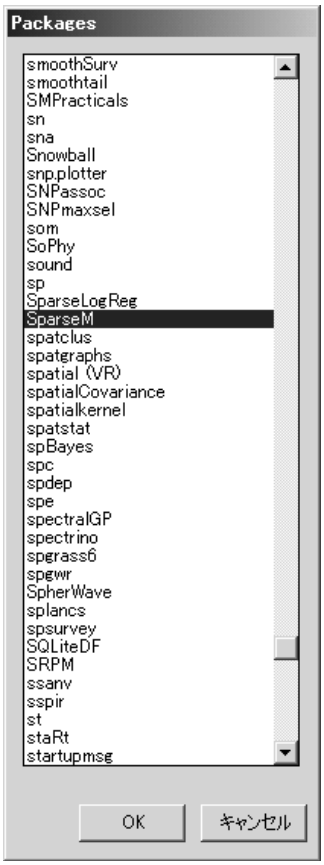

図 1.4: パッケージの選択

するとダウンロードが開始され、R のコンソールで以下のような表示がされ、パッケージの導入が完了します。

```
package 'SparseM' successfully unpacked and MD5 sums checked

The downloaded packages are in
        C:\Documents and Settings\shinnou\Local Settings
          \Temp\RtmpU03jsb\downloaded_packages
updating HTML package descriptions
```

パッケージ内の関数を利用するには、まずパッケージを読み込まないといけません。メニューバーの [パッケージ] → [パッケージの読み込み] からも読み込めますが、バッチ処

理することも考えると、コンソールから以下のように読み込む方法を知っておいた方がよいです。

```
> library(SparseM)
Package SparseM (0.73) loaded.  To cite, see citation("SparseM")
```

1.3　R プログラミングの基礎

1.3.1　条件分岐と繰り返し

プログラムを書く場合、条件分岐と繰り返しが基本です。
条件式の書式は以下です。

```
if (条件式) {
   式 11
   式 12
    ...
   式 1m
} else {
   式 21
   式 22
    ...
   式 2n
}
```

意味は明らかだと思います。条件式の結果が真であれば、式 11 から 式 1m が順に実行され、 偽であれば式 21 から 式 2n が順に実行されます。
式というのはここまでに説明したコマンドのことです。
また `else` の部分が必要ないときは省けます。

```
if (条件式) {
   式 11
   式 12
    ...
   式 1m
}
```

条件式には論理演算子を用いた式を書きます。論理値を扱う演算としては以下のものを押さえておけば十分です。

==	等しければ真
!=	等しくなれけば真
<	左辺が右辺よりも小さければ真
<=	左辺が右辺以下なら真
!	否定
&&	論理積
!!	論理和
xor	排他的論理和
TRUE または T	定数（真）
FALSE または F	定数（偽）

表 1.2: 論理を扱う演算子、関数、定数

if 式の例を以下に示します。

```
> (x0 <- runif(1,min=0,max=10)) # [0,10] の一様分布に従う乱数を 1 個作る。
[1] 9.405966
> (x1 <- floor(x0))
[1] 9
> if (x1 >= 5) { y <- 10; z <- 1 } else { y <- 1; z <- 2}
> y
[1] 10
> z
[1] 1
> if (floor(runif(1,min=0,max=10)) >= 5) { # このようにも書ける
+    y <- 10
+    z <- 1
+ } else {
+    y <- 1
+    z <- 2
+ }
> y     # 乱数なので、結果が違うこともある
[1] 1
> z
[1] 2
```

次に繰り返しの式ですが for と while を知っていれば十分です。
for の書式は以下です。

```
for(i in M) {
   式 11
   式 12
    ...
   式 1m
}
```

M はベクトルです。ベクトル M の要素を順に変数 i にバインドさせて、括弧内の式を実行します。
for 式の例を以下に示します。

```
> x <- 0
> for(i in c(1:100)) { x <- x + i }   # 1 から 100 までの和を求めてみます。
> x
[1] 5050
> sum(c(1:100))  # 上記の例は、普通、こうします
[1] 5050
```

また while の書式は以下です。

```
while(条件式) {
   式 11
   式 12
    ...
   式 1m
}
```

条件式が真である間、括弧内の式を繰り返します。
while 式の例を以下に示します。

```
> x <- 0; i <- 1
> while(i <= 100) { x <- x + i; i <- i + 1 }
> x
[1] 5050
> i
```

```
[1] 101
```

1.3.2 繰り返しの制御

繰り返し処理の中で、強制的に繰り返しを抜けたり、強制的に次の繰り返し処理に移りたい場合があります。C 言語での break と continue です。R では break と next を使います。例を示します。

```
> x <- 0
> for(i in c(1:100)) {
+    if (i >= 50) { break }   # i が 50 になるとループを抜ける
+    x <- x + i
+ }
> x
[1] 1225       # 結局、1 から 49 までの和が求まる
> x <- 0; i <- 1
> while(i <= 100) {
+    i <- i + 1
+    if (i %% 2 == 1) { next }  # i が奇数だと次の繰り返し処理へ
+    x <- x + i
+ }
> x
[1] 2550       # 結局、偶数だけの和が求まる
```

中括弧は複数の式をまとめるのに使います。中括弧の中の式が 1 つの場合、中括弧は省略できます。上記の例の該当箇所では、

```
if (i >= 50) break
```

や

```
if (i %% 2 == 1) next
```

と書けます。

1.3.3 関数の書き方

R でプログラムを書くとは関数を作ることを意味します。
関数は以下の書式で定義します。

```
関数名 <- function(引数 1, 引数 2, … , 引数 n) {
   式 11
   式 12
   …
   式 1m
}
```

例として第 1 引数のベクトルの最大値と第 2 引数のベクトルの最小値の和を求める関数 myfex() を作ってみます。

```
> myfex <- function(x,y) {
+    z <- max(x) + min(y)
+    z
+ }
> x <- c(1,2,-2,3)
> y <- c(0,-1,2,-2)
> w <- myfex(x,y)
> w
[1] 1
```

見て分かるとおり、全ての変数について型の宣言は必要ないので、記述は非常に簡単です。

関数内の変数は全て局所変数となることに気をつけて下さい。そのため関数を出た後に関数内の変数は参照できません。上記の例では、変数 z が局所変数なので、関数を出た後に変数 z は参照できません。

```
> z
エラー： オブジェクト "z" は存在しません
```

次に引数ですが、通常のプラグラム言語では構造体を渡すときに実質的にポインタを渡すので、引数となる構造体が破壊される可能性があります。R はここらを気にする必要ありません。関数内の変数は全て局所変数です。

また関数内で未定義の変数があった場合、その変数名をもつ大域変数が自動的に利用されます。例を示します。

```
> a <- 1
> myfex1 <- function(x,y) {
+    z <- max(x) + min(y) + a
+    z
```

```
+ }
> (w <- myfex1(x,y))
[1] 2
> a <- 2
> (w <- myfex1(x,y))
[1] 3
```

上記の関数 myfex1() 内の変数 a は未定義なので、関数の外で定義された a の値が使われています。

関数内から大域変数の値を変えたい場合には<<-という代入の記号を利用します。

```
> a <- 1
> myfex2 <- function(x,y) {
+     z <- max(x) + min(y) + a
+     a <<- a + 1
+     z
+ }
> (w <- myfex2(x,y))
[1] 2
> a
[1] 2
> (w <- myfex2(x,y))
[1] 3
> a
[1] 3
```

最後に、呼び出し側に関数の結果を返す方法ですが、関数内で最後に実行される式の値が返るので、返す値を関数が終了する箇所に書いておけば明確です。更にそれを return() に渡すのがよいとされています。

上記の例の関数 myfex() では、最終的に変数 z の値を返すので、最後に z という式を書きました。

最後の z を消して、その 1 つ前の

```
z <- max(x) + min(y)
```

という式で終わると、この式は代入式なので値が返りません。代入しないで、max(x) + min(y) を返り値にすればよいので、結局、以下のように書けます。

```
myfex <- function(x,y) { max(x) + min(y) }
```

これくらい単純なら、これでよいと思います。ただ返り値があるということを明確にするために return() を使う流儀もあります[3]。

```
myfex <- function(x,y) { return(max(x) + min(y)) }
```

また上記の例では返り値が数値でしたが、どのような構造のデータでも返り値にすることができます。

1.3.4　任意の引数とデフォルトの値

R で関数を記述する場合、任意の引数を設定することは容易です。

例えば、先ほどの myfex() に a という引数を増やして、以下のような関数を作ったとします。

```
myfex <- function(x,y,a) { max(x)**a + min(y)**a }
```

```
> myfex <- function(x,y,a) { max(x)**a + min(y)**a }
> myfex(x,y,1)
[1] 1
```

ここで a を任意の引数にして、省略された場合は a = 1 として処理するとします。この場合、以下のように関数を定義すればよいです。

```
> myfex <- function(x,y,a=1) { max(x)**a + min(y)**a }
> myfex(x,y)   ## 省略された場合は a = 1
[1] 1
> myfex(x,y,a=2) ## 任意の引数を設定するときは (変数名)=(値)
[1] 13
> myfex(x,y,2)  ## 順序があっていれば (変数名)= はいらない
[1] 13
```

1.3.5　ファイルのロード

一般にプログラム（関数群）はファイルに記述します。そのファイルをロードすることで、R 内でそのファイル内に記述された関数が使えるようになります。ファイルをロードするには関数 source() を利用します。

以下の内容のファイルが ex.r として、作業スペースのディレクトリに置かれているとします。

[3]私は関数が長いときは return() を使いますが、短い場合は使わないこともあります。本書のプログラムの記述でも、return() を使わないこともあります。

```
myfex <- function(x,y) {
   max(x) + min(y)
}
```

このとき以下のようにすることでファイル ex.r をロードでき、その中で定義した関数 myfex() が利用できるようになります。

```
> source("ex.r")   # ファイルをロード
> x <- c(1,2,-2,3)
> y <- c(0,-1,2,-2)
> (z <- myfex(x,y))
[1] 1
```

実は source() に与えるファイルの中身は、コマンド（式）の列です。関数の定義も式です。そのため上記の処理を全部ファイルに記述することも可能です。以下のファイル ex1.r を作ります。

```
myfex <- function(x,y) {
   max(x) + min(y)
}
x <- c(1,2,-2,3)
y <- c(0,-1,2,-2)
z <- myfex(x,y)
```

これを source() でロードしてみます。

```
> remove(z)    #  変数 z を取り除く
> z
 エラー： オブジェクト "z" は存在しません
> source("ex1.r")
> z            #  変数 z ができている
[1] 1
```

1.3.6 バッチ処理

R はバッチ処理も可能です。オプションがいろいろありますが、通常、以下で実行すればよいと思います。

```
C:\> R --vanilla -q < batch.r
```

この処理によってファイル batch.r に記述された R のコマンドが実行され、処理が終了すると R が終了します。また上記のオプションでは保存されている作業スペースなどは読み込みませんし、作業スペースも保存しません。

C:\> というのは DOS のプロンプトです。コマンド R.exe は C:\R\R-2.5.1\bin のディレクトリ下にあります。このディレクトリをサーチパスに含めていない場合は、絶対アドレスで R.exe を指定して下さい。

また batch.r では出力をファイルに保存する形にしていないと意味がありません。以下は batch.r の例です。

```
mysolve <- function(infile,outfile,nc) {
   x <- matrix(scan(infile), ncol=nc, byrow=TRUE)
                  # 行列作成
   ix <- solve(x)  # 逆行列
   write(t(ix), file=outfile,ncolumns=nc)  # 結果をファイルへ出力
}

mysolve("4x4.txt","kekka",4)   # 実行！
```

以下は実行例です。

```
C:\R\R-2.5.1>bin\R.exe --vanilla -q < batch.r
> mysolve <- function(infile,outfile,nc) {
+    x <- matrix(scan(infile), ncol=nc, byrow=TRUE)
+                     # 行列作成
+    ix <- solve(x)  # 逆行列
+    write(t(ix), file=outfile,ncolumns=nc)  # 結果をファイルへ出力
+ }
>
> mysolve("4x4.txt","kekka",4)   # 実行！
Read 16 items
>
C:\R\R-2.5.1>   # コマンドプロンプトに戻る
```

以下は 4x4.txt の中身と、作成されたファイル kekka の中身です。

4x4.txt の中身

```
1 2 0 1
0 1 1 2
1 0 1 1
```

```
1 3 1 0
```

kekka の中身

```
0.4444444 -0.5555556 0.6666667 -0.1111111
0.1111111 0.1111111 -0.3333333 0.2222222
-0.7777778 0.2222222 0.3333333 0.4444444
0.3333333 0.3333333 0 -0.3333333
```