S 语言介绍

By Li Dong Feng

http://www.ms.uky.edu/~mai/splus/chinese/HTML/Sintro.html

Li Dongfeng, Department of Probability and Statistics, Peking University, Beijing 100871, P.R. China, Email: ldf@math.pku.edu.cn, Homepage: http://www.math.pku.edu.cn/teachers/lidf/index.html

S 快速入门

背景介绍

S 语言是由 AT&T 贝尔实验室开发的一种用来进行数据探索、统计分析、作图的解释型语言。它的丰富的数据类型（向量、数组、列表、对象等）特别有利于实现新的统计算法，其交互式运行方式及强大的图形及交互图形功能使得我们可以方便的探索数据。

目前 S 语言的实现版本主要是 S-PLUS。它基于 S 语言，并由 MathSoft 公司的统计科学部进一步完善。作为统计学家及一般研究人员的通用方法工具箱，S-PLUS 强调演示图形、探索性数据分析、统计方法、开发新统计工具的计算方法，以及可扩展性。

S-PLUS 可以直接用来进行标准的统计分析得到所需结果，但是它的主要的特点是它可以交互地从各个方面去发现数据中的信息，并可以很容易地实现一个新的统计方法。

S-PLUS 有微机版本和工作站版本，它是一个商业软件。Auckland 大学的 Robert Gentleman 和 Ross Ihaka 及其他志愿人员开发了一个 R 系统，其语法形式与 S 语言基本相同，但实现不同，两种语言的程序有一定的兼容性。R 是一个 GPL 自由软件，现在的版本是 1.00 版，它比 S-PLUS 还少许多功能，但已经具有了很强的实用性。我们在这里尽量介绍 S-PLUS 和 R 都能使用的功能，且以 R 为主。下面我们用 S 统称 S-PLUS 和 R。

入门实例

S 的基本界面是一个交互式命令窗口，命令提示符是一个大于号，命令的结果马上显示在命令下面。S 命令主要有两种形式：表达式或赋值运算 （用<－表示）。在命令提示符后键入一个表达式表示计算此表达式并显示结果。赋值运算把赋值号右边的值计算出来赋给左边的变量。可以用向上光标键来 找回以前运行的命令再次运行或修改后再运行。

S 是区分大小写的，所以 x 和 X 是不同的名字。

我们用一些例子来看 S-PLUS 的特点。假设我们已经进入了 S-PLUS（或 R）的交互式窗口。如果没有打开的图形窗口，在 R 中，用：

> x11()

在 S-PLUS Windows 版中用：

> win.graph()

可以打开一个作图窗口。然后，输入以下语句：

```
> x1 <- 0:100
> x2 <- x1*2*pi/100
> y <- sin(x2)
> plot(x,y, type='l')
```

这些语句可以绘制正弦曲线图。其中，“<－”是赋值运算符。0:100 表示一个从 0 到 100 的等差数列向量。从第二个语句可以看出，我们可以对向量直接进行四则运算，计算得到的 x2 是向量 x1 的所有元素乘以常数 2*pi/100 的结果。从第三个语句可以看到函数可以以向量为输入，并可以输出一个向量，结果向量 y 的每一个分量是自变量 x2 的每一个分量的正弦函数值。从最后一个语句可以看出函数的调用也很自由，可以按位置给出自变量，也可以用“自变量名＝”的形式指定自变量值，这样可以 使用缺省值。

下面我们看一看 S 的统计功能。

```
> marks <- c(10, 6, 4, 7, 8)
> mean(marks)
[1] 7
> sd(marks)
[1] 2.236068
> median(marks)
[1] 7
> min(marks)
[1] 4
> max(marks)
[1] 10
> boxplot(marks)
>
```

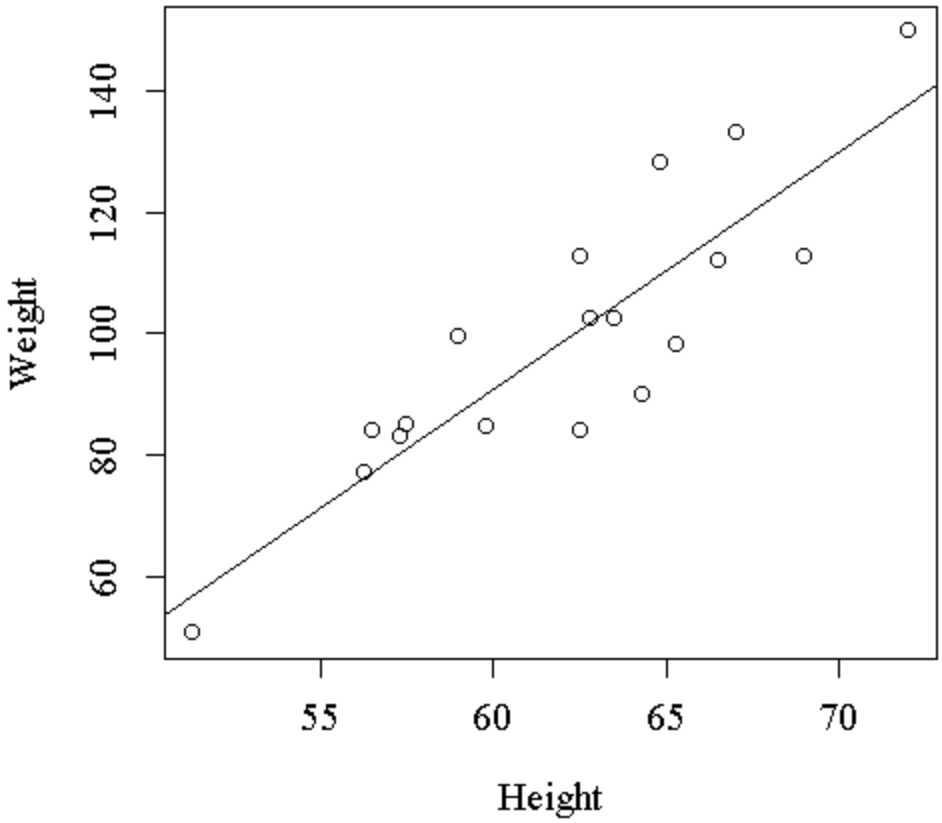
第一个语句输入若干数据到一个向量，函数 c()用来把数据组合为一个向量。后面用了几个函数来计算数据的均值、标准差、中位数、最小值、最大值。最 后的函数绘制数据的盒形图。例中 sd()是 R 中才有的函数，在 S-PLUS 中要用 sqrt(var()) 来计算。在 S 命令方式中要显示一个表达式的值只要 键入它。

为了演示 S 的回归计算，我们把 SAS 中的例子数据 SASUSER.CLASS 输出到了一个文本文件 CLASS.TXT，SAS 程序如下：

```
data _null_;
  set sasuser.class;
  file 'class.txt';
  put name sex age height weight;
run;
```

我们在 R 中把这个文件读入为为一个数据框（data frame，相当于 SAS 中的数据集），并进行回归，绘制数据散点图和回归直线。假设图形窗口已经打开，程序及结果如下：

```
> cl <- read.table("c:/work/class.txt",
+     col.names=c("Name", "Sex", "Age", "Height", "Weight"))
> cl
```




```
      Name Sex Age Height Weight
1    Alice   F  13   56.5   84.0
2    Becka   F  13   65.3   98.0
3     Gail   F  14   64.3   90.0
4    Karen   F  12   56.3   77.0
5    Kathy   F  12   59.8   84.5
6     Mary   F  15   66.5  112.0
7    Sandy   F  11   51.3   50.5
8   Sharon   F  15   62.5  112.5
9    Tammy   F  14   62.8  102.5
10  Alfred   M  14   69.0  112.5
11    Duke   M  14   63.5  102.5
12   Guido   M  15   67.0  133.0
13   James   M  12   57.3   83.0
14 Jeffrey   M  13   62.5   84.0
15    John   M  12   59.0   99.5
16  Philip   M  16   72.0  150.0
17  Robert   M  12   64.8  128.0
18  Thomas   M  11   57.5   85.0
19 William   M  15   66.5  112.0
> attach(cl)
> plot(Height, Weight)
```

```
> fit1 <- lm(Weight ~ Height)
> fit1

Call:
lm(formula = Weight ~ Height)

Coefficients:
(Intercept)      Height
 -143.02692     3.89903

> abline(fit1)
>
```

结果可以产生图 1。我们从这个例子可以看到，分析由函数完成，结果也是对象，可以作为进一步分析的输入。

S-PLUS 也提供了一般的计算功能。比如，求一个矩阵的逆：

```
> A <- matrix(c(1,2, 7,3), ncol=2, byrow=T)
> A
     [,1] [,2]
[1,]    1    2
[2,]    7    3
> Ai <- solve(A)
> Ai
           [,1]        [,2]
[1,] -0.2727273  0.18181818
[2,]  0.6363636 -0.09090909
```

还可以进行矩阵运算，如：

```
> b <- c(2,3)
> x <- Ai %*% b
> x
              [,1]
[1,]-2.220446e-016
[2,] 1.000000e+000
>
```

这实际上是解了一个线性方程组。“%*%”表示矩阵乘法。

可以把若干行命令保存在一个文本文件（比如 C:\WORK\MYPROG.S）中，然后用 source 函数来运行整个文件：

> source("C:\\WORK\\MYPROG.S")

注意字符串中的反斜杠要写成两个。用 sink()函数可以把以后的输出从屏幕窗口转向到一个外部文本文件，例如：

> sink("C:\\WORK\\MYPROG.OUT")

要恢复输出到屏幕窗口，使用：

> sink()

在 S 中可以用“?”号后面跟要查询的函数来显示帮助信息，例如

> ?c

> ?"=="

（查询特殊符号要用撇号括起来）。另外，在 MS Windows 版的 S-PLUS 中可以用 Windows 帮助来得到帮助，而在 R 中可以用 HTML 浏览器来查询帮助（用“help.start()函数或 Help | R language (html)”启动）。用好 S 语言重要的一点就是要用熟系统的帮助功能，因为 S 的教材或手册都不能列出所有的细节，如果列出所有细节则过于繁琐了。

要退出 S-PLUS 或 R，可以用 q()函数，也可以用菜单命令。R 在退出时提问是否保存当前工作空间，它可以把当前定义的所有对象（有名字的向量、矩阵、列表、函数等）保存到一个文件。S-PLUS 自动把对象保存到一个子目录，子目录中每一个文件保存一个对象。

S 向量

S 语言是基于对象的语言，不过它的最基本的数据还是一些类型，如向量、矩阵、列表（list）等。更复杂的数据用对象表示，比如，数据框对象，时间序列对象，模型对象，图形对象，等等。

S 语言表达式可以使用常量和变量。变量名的规则是：由字母、数字、句点组成，第一个字符必须是字母，长度没有限制。大小写是不同的。特别要注意句点 可以作为名字的合法部分，而在其它面向对象语言中句点经常用来分隔对象与成员名。另外，下划线不能用在名字中，因为它是赋值符号“<—”的缩写。

常量

常量可以笼统地分为逻辑型、数值型和字符型三种，实际上数值型数据又可以分为整型、单精度、双精度等，非特殊需要不必太关心其具体类型。例如，123，123.45，1.2345e30 是数值型常量，"Weight"，"李明"是字符型（用两个双撇号或两

个单撇号包围）。逻辑真值写为 T 或 TRUE（注意区分大小写，写 t 或 true 都没意义），逻辑假值写为 F 或 FALSE。复数常量就用 3.5-2.1i 这样的写法表示。

S 中的数据可以取缺失值，用符号 NA 代表缺失值。函数 is.na(x)返回 x 是否缺失值（真还是假）。

向量（Vector）与赋值

向量是具有相同基本类型的元素序列，大体相当于其他语言中的一维数组。实际上在 S 中标量也被看作是长度为 1 的向量。

定义向量的最常用办法是使用函数 c()，它把若干个数值或字符串组合为一个向量，比如：

```
> marks <- c(10, 6, 4, 7, 8)
> x <- c(1:3, 10:13)
> x
[1]  1  2  3 10 11 12 13
> x1 <- c(1, 2)
> x2 <- c(3, 4)
> x <- c(x1, x2)
> x
[1] 1 2 3 4
```

在显示向量值时我们注意到左边总出现一个“[1]”，这是代表该显示行的第一个数的下标，例如：

> 1234501:1234520

[1] 1234501 1234502 1234503 1234504 1234505 1234506 1234507 1234508 1234509 1234510

[11] 1234511 1234512 1234513 1234514 1234515 1234516 1234517 1234518 1234519 1234520

第二行输出从第 11 个数开始，所以在行左边显示“[11]”。

S 中用符号“<—”（这是小于号紧接一个减号）来为变量赋值。另一种赋值的办法是用 assign 函数，比如

```
> x1 <- c(1, 2)
```

和

```
> assign("x1", c(1, 2))
```

效果相同。

函数 length(x)可以计算向量 x 的长度。

向量运算

可以对向量进行加（＋）减（－）乘（*）除（/）、乘方（^）运算，其含意是对向量的每一个元素进行运算。；例如：

```
> x <- c(1, 4, 6.25)
> y <- x*2+1
> y
[1]  3.0  9.0 13.5
```

另外，%/%表示整数除法（比如 5 %/% 3 为 1），%%表示求余数（如 5 %% 3 为 2）。

也可以用向量作为函数自变量，sqrt、log、exp、sin、cos、tan 等函数都可以用向量作自变量，结果是对向量的每一个元素取相应的函数值，如：

```
> sqrt(x)
[1] 1.0 2.0 2.5
```

函数 min 和 max 分别取自变量向量的最小值和最大值，函数 sum 计算自变量向量的元素和，函数 mean 计算均值，函数 var 计算样本方差，函数 sd 计算标准差（在 Splus 中用 sqrt(var())计算），函数 range 返回包含两个值的向量，第一个值是最小值，第二个值是最大值。例如：

```
> max(x)
[1] 6.25
```

如果求 var(x)而 x 是 $n \times p$ 矩阵，则结果为样本协方差阵。

sort(x)返回 x 的元素从小到大排序的结果向量。order(x)返回使得 x 从小到大排列的元素下标向量（x[order(x)]等效于 order(x)）。

任何数与缺失值的运算结果仍为缺失值。例如，

```
>  2*c(1, NA, 2)
[1]  2 NA  4
> sum(c(1, NA, 2))
[1] NA
```

产生有规律的数列

在 S 中很容易产生一个等差数列。例如，1:n 产生从 1 到 n 的整数列，-2:3 产生从-2 到 3 的整数列，5:2 产生反向的数列：

```
> n <- 5
```

```
> 1:n
[1] 1 2 3 4 5
> -2:3
[1] -2 -1  0  1  2  3
> 5:2
[1] 5 4 3 2
```
要注意 1:n-1 不是代表 1 到 n-1 而是向量 1:n 减去 1，这是一个常犯的错误：
```
> 1:n-1
[1] 0 1 2 3 4
> 1:(n-1)
[1] 1 2 3 4
```

seq 函数是更一般的等差数列函数。如果只指定一个自变量 n>0，则 seq(n)相当于 1:n 。指定两个自变量时，第一自变量是开始值，第二自变量是结束值，如 seq(-2,3)是从-2 到 3 。S 函数调用的一个很好的特点是它可以使用不同个数的自变量，函数可以对不同类型的自变量给出不同结果，自变量可以用“自变量名＝自变量值”的形式指定。例如，seq(-2,3)可以写成 seq(from=-2, to=3)。可以用一个 by 参数指定等差数列的增加值，例如：

```
> seq(0, 2, 0.7)
[1] 0.0 0.7 1.4
```
也可以写成 seq(from=0, to=2, by=0.7)。有参数名的参数的次序任意，如：
```
> seq(0, by=0.7, to=2)
[1] 0.0 0.7 1.4
```

可以用 length 参数指定数列长度，如 seq(from=10, length=5)产生 10 到 14。seq 函数还可以用一种 seq(along=向量名)的格式，这时只能用这一个参数，产生该向量的下标序列，如：

```
> x
[1] 1.00 4.00 6.25
> seq(along=x)
[1] 1 2 3
```
另一个类似的函数是 rep，它可以重复第一个自变量若干次，例如：
```
> rep(x,3)
[1] 1.00 4.00 6.25 1.00 4.00 6.25 1.00 4.00 6.25
```
第一个参数名为 x，第二个参数名为 times（重复次数）。

逻辑向量

向量可以取逻辑值，如：

```
> l <- c(T, T, F)
> l
[1]  TRUE  TRUE FALSE
```
当然，逻辑向量是一个比较的结果，如：

```
> x
[1] 1.00 4.00 6.25
```

```
> l <- x > 3
> l
[1] FALSE  TRUE  TRUE
```
一个向量与常量比较大小，结果还是一个向量，元素为每一对比较的结果逻辑值。两个向量也可以比较，如：

```
> log(10*x)
[1] 2.302585 3.688879 4.135167
> log(10*x) > x
[1]  TRUE FALSE FALSE
```
比较运算符包括<，<=，>，>=，==（相等），!=（不等）。

两个逻辑向量可以进行与（&）、或（|）运算，结果是对应元素运算的结果。对逻辑向量 x 计算!x 表示取每个元素的非。例如：

```
> (x > 1.5) & log(10*x)>1.5
[1] FALSE FALSE  TRUE
```
判断一个逻辑向量是否都为真值的函数是 all，如：

```
> all(log(10*x) > x)
[1] FALSE
```
判断是否其中有真值的函数是 any，如：

```
> any(log(10*x) > x)
[1] TRUE
```
函数 is.na(x)用来判断 x 的每一个元素是否缺失。如

```
> is.na(c(1, NA, 3))
[1] FALSE  TRUE FALSE
```

&# 9;逻辑值可以强制转换为整数值，TRUE 变成 1，FALSE 变成 0。例如，我们以 age>65 为老年人，否则为年轻人，可以用 c("young", "old")[(age>65)+1]这样的表示。当年龄大于 65 时 age>65 等于 TRUE，加 1 则把 TRUE 转换为数值型的 1，结果得 2，于是返回第二个下标处的“old”。否则等于 0+1 下标处的“young”。

字符型向量

向量元素可以取字符串值。例如：

```
> c1 <- c("x", "sin(x)")
> c1
[1] "x"      "sin(x)"
> ns <- c("Weight", "Height", "年龄")
> ns
[1] "Weight" "Height" "年龄"
```
paste 函数用来把它的自变量连成一个字符串，中间用空格分开，例如：
```
> paste("My", "Job")
[1] "My Job"
```

连接的自变量可以是向量，这时各对应元素连接起来，长度不相同时较短的向量被重复使用。自变量可以是数值向量，连接时自动转换成适当的字符串表示，例如：

```
> paste(c("X", "Y"), "=", 1:4)
[1] "X = 1" "Y = 2" "X = 3" "Y = 4"
```

分隔用的字符可以用 sep 参数指定，例如下例产生若干个文件名：

```
> paste("result.", 1:6, sep="")
[1] "result.1" "result.2" "result.3" "result.4" "result.5" "result.6"
```

如果给 paste()函数指定了 collapse 参数，则 paste()可以把一个字符串向量的各个元素连接成一个字符串，中间用 collapse 指定的值分隔。比如 paste(c('a', 'b'), collapse='.') 得到'a.b'。

复数向量

S 支持复数运算。复数常量只要用 3.5+2.1i 这样的格式即可。复向量的每一个元素都是复数。可以用 complex()函数生成复向量（见帮助）。Re()计算实部，Im()计算虚部，Mod() 计算复数模，Arg()计算复数幅角。

向量下标运算

S 提供了十分灵活的访问向量元素和向量子集的功能。某一个元素只要用 x[i]的格式访问，其中 x 是一个向量名，或一个取向量值的表达式，如：

```
> x
[1] 1.00 4.00 6.25
> x[2]
[1] 4
> (c(1, 3, 5) + 5)[2]
```

[1] 8

可以单独改变一个元素的值，例如：

```
> x[2] <- 125
> x
[1]   1.00 125.00   6.25
```

事实上，S 提供了四种方法来访问向量的一部分，格式为 x[v]，x 为向量或向量值的表达式，v 是如下的表示下标向量：

一、取正整数值的下标向量

v 为一个向量，取值在 1 到 length(x)之间，取值允许重复，例如，

```
> x[c(1,3)]
[1] 1.00 6.25
> x[1:2]
[1]   1 125
> x[c(1,3,2,1)]
[1]   1.00   6.25 125.00   1.00
> c("a", "b", "c")[rep(c(2,1,3), 3)]
[1] "b" "a" "c" "b" "a" "c" "b" "a" "c"
```

二、取负整数值的下标向量

v 为一个向量，取值在－length(x)到－1 之间，表示扣除相应位置的元素。例如：

> x[-(1:2)]

[1] 6.25

三、取逻辑值的下标向量

v 为和 x 等长的逻辑向量，x[v]表示取出所有 v 为真值的元素，如：

```
> x
[1]   1.00 125.00   6.25
> x<10
[1]  TRUE FALSE  TRUE
> x[x<10]
[1] 1.00 6.25
> x[x<0]
numeric(0)
```

可见 x[x<10]取出所有小于 10 的元素组成的子集。这种逻辑值下标是一种强有力的检索工具，例如 x[sin(x)>0]可以取出 x 中所有正弦函数值为正的元素组成的向量。

如果下标都是假值则结果是一个零长度的向量，显示为 numeric(0)。

四、取字符型值的下标向量

在定义向量时可以给元素加上名字，例如：

```
> ages <- c(Li=33, Zhang=29, Liu=18)
> ages
   Li Zhang   Liu
   33    29    18
```

这样定义的向量可以用通常的办法访问，另外还可以用元素名字来访问元素或元素子集，例如：

```
> ages["Zhang"]
Zhang
   29
> ages[c("Li", "Liu")]
 Li Liu
 33  18
```

向量元素名可以后加，例如：

```
> ages1 <- c(33, 29, 18)
> names(ages1) <- c("Li", "Zhang", "Liu")
> ages1
   Li Zhang   Liu
   33    29    18
```

上面我们看到了如何访问向量的部分元素。在 S 中还可以改变一部分元素的值，例如：

```
> x
[1]   1.00 125.00   6.25
> x[c(1,3)] <- c(144, 169)
> x
[1] 144 125 169
```

注意赋值的长度必须相同，例外是可以把部分元素赋为一个统一值：

```
> x[c(1,3)] <- 0
> x
[1]   0 125   0
```

要把向量所有元素赋为一个相同的值而又不想改变其长度，可以用 x[]的写法：

> x[] <- 0

注意这与“x <- 0”是不同的，前者赋值后向量长度不变，后者使向量变为标量 0。

改变部分元素值的技术与逻辑值下标方法结合可以定义向量的分段函数，例如，要定义 y=f(x)为当 x<0 时取 1-x，否则取 1+x，可以用：

```
> y <- numeric(length(x))
> y[x<0] <- 1 - x[x<0]
> y[x>=0] <- 1 + x[x>0]
```

多维数组和矩阵

数组（array）和矩阵(matrix)

数组（array）可以看成是带多个下标的类型相同的元素的集合，常用的是数值型的数组如矩阵，也可以有其它类型（如字符型、逻辑型、复型数组）。S 可以很容易地生成和处理数组，特别是矩阵（二维数组）。

数组有一个特征属性叫做维数向量（dim 属性），维数向量是一个元素取正整数值的向量，其长度是数组的维数，比如维数向量有两个元素时数组为二维数组（矩阵）。维数向量的每一个元素指定了该下标的上界，下标的下界总为 1。

一组值只有定义了维数向量（dim 属性）后才能被看作是数组。比如：

```
> z <- 1:1500
> dim(z) <- c(3, 5, 100)
```

这时 z 已经成为了一个维数向量为 c(3,5,100)的三维数组。也可以把向量定义为一维数组，例如：

> dim(z) <- 1500

数组元素的排列次序缺省情况下是采用 FORTRAN 的数组元素次序（按列次序），即第一下标变化最快，最后下标变化最慢，对于矩阵（二维数组）则是按列存放。例如，假设数组 a 的元素为 1:24，维数向量为 c(2,3,4)，则各元素次序为 a[1,1,1], a[2,1,1], a[1,2,1], a[2,2,1], a[1,3,1], ..., a[2,3,4]。

用函数 array()或 matrix()可以更直观地定义数组。array()函数的完全使用为 array(x, dim=length(x), dimnames=NULL)，其中 x 是第一自变量，应该是一个向量，表示数组的元素值组成的向量。dim 参数可省，省略时作为一维数组（但不同于向 量）。dimnames 属性可以省略，不省略时是一个长度与维数相同的列表（list，见后面），列表的每个成员为一维的名字。例如上面的 z 可以这样定 义：

```
> z <- array(1:1500, dim=c(3,5,100))
```

函数 matrix()用来定义最常用的一种数组：二维数组，即矩阵。其完全格式为

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

其中第一自变量 data 为数组的数据向量（缺省值为缺失值 NA），nrow 为行数，ncol 为列数，byrow 表示数据填入矩阵时按行次序还是列次 序，一定注意缺省情况下按列次序，这与我们写矩阵的习惯是不同的。dimnames 缺省是空值，否则是一个长度为 2 的列表，列表第一个成员是长度与行数相 等的字符型向量，表示每行的标签，列表第二个成员是长度与列数相同的字符型向量，表示每列的标签。例如，定义一个 3 行 4 列，由 1:12 按行次序排列的矩 阵，可以用：

```
> b <- matrix(1:12, ncol=4, byrow=T)
> b
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

注意在有数据的情况下只需指定行数或列数之一。指定的数据个数允许少于所需的数据个数，这时循环使用提供的数据。例如：

```
> b <- matrix(0, nrow=3, ncol=4)
```

生成 3 行 4 列的元素都为 0 的矩阵。

数组下标

要访问数组的某个元素，只要象上面那样写出数组名和方括号内用逗号分开的下标即可，如 a[2,1,2]。

更进一步我们还可以在每一个下标位置写一个下标向量，表示对这一维取出所有指定下标的元素，如 a[1, 2:3, 2:3]取出所有第一下标为 1，第二下标为 2 或 3，第三下标为 2 或 3 的元素。注意因为第一维只有一个下标所以退化了，得到的是一个维数向量为 c(2,2)的数组。

另外，如果略写某一维的下标，则表示该维全选。例如，a[1, ,]取出所有第一下标为 1 的元素，得到一个形状为 c(3,4)的数组。a[, 2,]取出所有第二下标为 2 的元素得到一个形状为 c(2,4)的数组。a[1,1,]则只能得到一个长度为 4 的向量，不再是数组（dim(a[1,1,]) 值为 NULL）。a[, ,]或 a[]都表示整个数组。比如

```
a[] <- 0
```

可以在不改变数组维数的条件下把元素都赋成 0。

还有一种特殊下标是对于数组只用一个下标向量（是向量，不是数组），比如 a[3:4] ，这时忽略数组的维数信息，把下标表达式看作是对数组的数据向量取子集。

不规则数组下标

在 S 中甚至可以把数组中的任意位置的元素作为一组访问，其方法是用一个矩阵作为数组的下标，矩阵的每一行是一个元素的下标，数组有几维下标矩阵的每一行就有几列。例如，我们要把上面的形状为 c(2,3,4)的数组 a 的第[1,1,1], [2,2,3], [1,3,4], [2,1,4]号共四个元素作为一个整体访问，先定义一个包含这些下标作为行的二维数组:

```
> b <- matrix(c(1,1,1, 2,2,3, 1,3,4, 2,1,4), ncol=3, byrow=T)
> b
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    3
[3,]    1    3    4
[4,]    2    1    4
> a[b]
[1]  1 16 23 20
```

注意取出的是一个向量。我们还可以对这几个元素赋值，如：

```
> a[b] <- c(101, 102, 103, 104)
> a
```

或

```
> a[b] <- 0
> a
```

数组四则运算

数组可以进行四则运算（+，—, *, /，^），解释为数组对应元素的四则运算，参加运算的数组一般应该是相同形状的（dim 属性完全相同）。例如，假设 A, B, C 是三个形状相同的数组，则

> D <- C + 2*A/B

计算得到的结果是 A 的每一个元素除以 B 的对应元素加上 C 的对应元素乘以 2 得到相同形状的数组。四则运算遵循通常的优先级规则。

形状不一致的向量和数组也可以进行四则运算，一般的规则是数组的数据向量对应元素进行运算，把短的循环使用来与长的匹配，并尽可能保留共同的数组属性。例如：

```
> x1 <- c(100, 200)
> x2 <- 1:6
> x1+x2
[1] 101 202 103 204 105 206
> x3 <- matrix(1:6, nrow=3)
> x3
     [,1] [,2]
```

```
[1,]    1    4
[2,]    2    5
[3,]    3    6
> x1+x3
     [,1] [,2]
[1,]  101  204
[2,]  202  105
[3,]  103  206
```

除非你清楚地知道规则应避免使用这样的办法（标量与数组或向量的四则运算除外）。

矩阵运算

矩阵是二维数组，但因为其应用广泛所以对它定义了一些特殊的运算和操作。

函数 t(A)返回矩阵 A 的转置。nrow(A)为矩阵 A 的行数，ncol(A)为矩阵 A 的列数。

矩阵之间进行普通的加减乘除四则运算仍遵从一般的数组四则运算规则，即数组的对应元素之间进行运算，所以注意 A*B 不是矩阵乘法而是矩阵对应元素相乘。

要进行矩阵乘法，使用运算符%*%，A%*%B 表示矩阵 A 乘以矩阵 B（当然要求 A 的列数等于 B 的行数）。例如：

```
> A <- matrix(1:12, nrow=4, ncol=3, byrow=T)
> B <- matrix(c(1,0), nrow=3, ncol=2, byrow=T)
> A
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
> B
     [,1] [,2]
[1,]    1    0
[2,]    1    0
[3,]    1    0
> A %*% B
     [,1] [,2]
[1,]    6    0
[2,]   15    0
[3,]   24    0
[4,]   33    0
>
```

另外，向量用在矩阵乘法中可以作为行向量看待也可以作为列向量看待，这要看哪一种观点能够进行矩阵乘法运算。例如，设 x 是一个长度为 n 的向量，A 是一个 $n \times n$ 矩阵，则“x %*% A %*% x”表示二次型 $x'Ax$ 。但是，有时向量在矩阵乘法中的地位并不清楚，比如“x %*% x”就既可能表示内积 $x'x$ 也可能表示 $n \times n$ 阵

xx'。因为前者较常用，所以 S 选择表示前者，但内积最好还是用 crossprod(x)来计算。要表示 xx'，可以用“cbind(x) %*% x”或“x %*% rbind(x) ”。

函数 crossprod(X, Y)表示一般的交叉乘积（内积） $X'Y$，即 X 的每一列与 Y 的每一列的内积组成的矩阵。如果 X 和 Y 都是向量则是一般的内积。只写一个参数 X 的 crossprod(X)计算 X 自身的内积 $X'X$。

其它矩阵运算还有 solve(A,b)解线性方程组 $Ax = b$，solve(A)求方阵 A 的逆矩阵，svd()计算奇异值分解，qr()计算 QR 分解，eigen()计算特征向量和特征值。详见随机帮助，例如：

> ?qr

函数 diag()的作用依赖于其自变量。diag(vector)返回以自变量（向量）为主对角元素的对角矩阵。diag(matrix)返回由矩阵的主对角元素组成的向量。diag(k)（k 为标量）返回 k 阶单位阵。

矩阵合并与拉直

函数 cbind()把其自变量横向拼成一个大矩阵，rbind()把其自变量纵向拼成一个大矩阵。cbind()的自变量是矩阵或者看作列向量的向 量，自变量的高度应该相等（对于向量，高度即长度，对于矩阵，高度即行数）。rbind 的自变量是矩阵或看作行向量的向量，自变量的宽度应该相等（对于向 量，宽度即长度，对于矩阵，宽度即列数）。如果参与合并的自变量比其它自变量短则循环补足后合并。例如：

```
> x1 <- rbind(c(1,2), c(3,4))
> x1
     [,1] [,2]
[1,]    1    2
[2,]    3    4
> x2 <- 10+x1
> x3 <- cbind(x1, x2)
> x3
     [,1] [,2] [,3] [,4]
[1,]    1    2   11   12
[2,]    3    4   13   14
> x4 <- rbind(x1, x2)
> x4
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]   11   12
[4,]   13   14
> cbind(1, x1)
     [,1] [,2] [,3]
[1,]    1    1    2
```

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

因为 cbind()和 rbind()的结果总是矩阵类型（有 dim 属性且为二维），所以可以用它们把向量表示为 $n \times 1$ 矩阵（用 cbind(x)）或 $1 \times n$ 矩阵（用 rbind(x)）。

设 a 是一个数组，要把它转化为向量（去掉 dim 和 dimnames 属性），只要用函数 as.vector(a) 返回值就可以了（注意函数只能通过函数值返回结果而不允许修改它的自变量，比如 t(X)返回 X 的转置矩阵而 X 本身并未改变）。另一种由数组得到其数据向量 的简单办法是使用函数 c()，例如 c(a)的结果是 a 的数据向量。这样的另一个好处是 c()允许多个自变量，它可以把多个自变量都看成数据向量连接起来。例如，设 A 和 B 是两个矩阵， 则 c(A,B)表示把 A 按列次序拉直为向量并与把 B 按列次序拉直为向量的结果连接起来。一定注意拉直时是按列次序拉直的。

数组的维名字

数组可以有一个属性 dimnames 保存各维的各个下标的名字，缺省时为 NULL（即无此属性）。我们以矩阵为例，它有两个维：行维和列维。比如， 设 x 为 2 行 3 列矩阵，它的行维可以定义一个长度为 2 的字符向量作为每行的名字，它的列维可以定义一个长度为 3 的向量作为每列的名字，属性 dimnames 是一个列表，列表的每个成员是一个维名字的字符向量或 NULL。例如：

```
> x <- matrix(1:6, ncol=2,
+ dimnames=list(c("one", "two", "three"), c("First", "Second")),
+ byrow=T)
> x
      First Second
one       1      2
two       3      4
three     5      6
```

我们也可以先定义矩阵 x 然后再为 dimnames(x)赋值。

对于矩阵，我们还可以使用属性 rownames 和 colnames 来访问行名和列名。如：

```
> x <- matrix(1:6, ncol=2,byrow=T)
> colnames(x) <- c("First", "Second")
> rownames(x) <- c("one", "two", "three")
```

在定义了数组的维名后我们对这一维的下标就可以用它的名字来访问，例如：

```
> x[c("one","three"),]
      First Second
one       1      2
three     5      6
```

数组的外积

两个数组 a 和 b 的外积是由 a 的每一个元素与 b 的每一个元素搭配在一起相乘得到一个新元素，这样得到一个维数向量等于 a 的维数向量与 b 的维数向量连起来的数组，即若 d 为 a 和 b 的外积，则 dim(d)=c(dim(a), dim(b))。

a 和 b 的外积记作 a %o% b。如

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

也可以写成一个函数调用的形式：

```
> d <- outer(a, b, '*')
```

注意 outer(a, b, f)是一个一般性的外积函数，它可以把 a 的任一个元素与 b 的任意一个元素搭配起来作为 f 的自变量计算得到新的元素值，外积是两个元素相乘的情况。函数当然也可以是加、减、除，或其它一般函数。当函数为乘积时可以省略不写。

例如，我们希望计算函数 $z = \cos(y)/(1+x^2)$ 在一个 x 和 y 的网格上的值用来绘制三维曲面图，可以用如下方法生成网格及函数值：

```
> x <- seq(-2, 2, length=20)
> y <- seq(-pi, pi, length=20)
> f <- function(x, y) cos(y)/(1+x^2)
> z <- outer(x, y, f)
```

用这个一般函数可以很容易地把两个数组的所有元素都两两组合一遍进行指定的运算。

下面考虑一个有意思的例子。我们考虑简单的 2×2 矩阵 $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$，其元素均在 0,1,...,9 中取值。假设四个元素 *a, b, c, d* 都是相互独立的服从离散均匀分布的随机变量，我们设法求矩阵行列式 *ad-bc* 的分布。首先，随机变量 *ad* 和 *bc* 同分布，它的取值由以下外积矩阵给出，每一个取值的概率均为 1/100:

```
> d <- outer(0:9, 0:9)
```

这个语句产生一个 10×10 的外积矩阵。为了计算 *ad* 的 100 个值（有重复）与 *bc* 的 100 个值相减得到的 10000 个结果，可以使用如下外积函数：

```
> d2 <- outer(d, d, "-")
```

这样得到一个维数向量为 c(2,2,2,2)的四维数组，每一个元素为行列式的一个可能取值，概率为万分之一。因为这些取值中有很多重复，我们可以用一个 table()函数来计算每一个值的出现次数（频数）：

```
> fr <- table(d2)
```

得到的结果是一个带有元素名的向量 fr，fr 的元素名为 d2 的一个取值，fr 的元素值为 d2 该取值出现的频数，比如 fr[1]的元素名为－81，值 为 19，表示值－81 在

数组 d2 中出现了 19 次。通过计算 length(fr)可以知道共有 163 个不同值。还可以把这些值绘制一个频数分布图（除以 10000 则为实际概率）：

```
> plot(as.numeric(names(fr)), fr, type="h",
+   xlab="行列式", ylab="频数")
```

其中 as.numeric()把向量 fr 中的元素名又转换成了数值型，用来作为作图的横轴坐标，fr 中的元素值即频数作为纵轴，type="h"表示是画垂线型图。

数组的广义转置

可以用 aperm(a, perm)函数把数组 a 的各维按 perm 中指定的新次序重新排列。例如：

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

结果 a 的第 2 维变成了 b 的第 1 维，a 的第 3 维变成了 b 的第 2 维，a 的第 1 维变成了 b 的第 3 维。这时有 a[i1,i2,i3]≡b[i2,i3,i1]。注意 c(i1,i2,i3)[2,3,1]=c(i2,i3,i1)。一般地，若 b <- aperm(a, p)，i 是数组 a 的一个下标向量，则 a[rbind(i)]≡b[rbind(i[p])]，即 a 的一个元素下标经过 p 变换后成为 b 的对应元素的下标。

对于矩阵 a，aperm(a, c(2,1))恰好是矩阵转置。对于矩阵转置可以简单地用 t(a)表示。

apply 函数

对于向量，我们有 sum、mean 等函数对其进行计算。对于数组，如果我们想对其中一维（或若干维）进行某种计算，可以用 apply 函数。其一般形式为：

apply(X, MARGIN, FUN, ...)

其中 X 为一个数组，MARGIN 是固定哪些维不变，FUN 是用来计算的函数。例如，设 a 是 3×4 矩阵，则 apply(a, 1, sum)的意义是对 a 的各行求和（保留第一维即第一个下标不变），结果是一个长度为 3 的向量（与第一维长度相同），而 apply(a, 2, sum)意义是对 a 的各列求和，结果是一个长度为 4 的向量（与第二维长度相同）。

如果函数 FUN 的结果是一个标量，MARGIN 只有一个元素，则 apply 的结果是一个向量，其长度等于 MARGIN 指定维的长度，相当于固定 MARGIN 指定的那一维的每一个值而把其它维取出作为子数组或向量送入 FUN 中进行运算。如果 MARGIN 指定了多个维，则结果是一个维数向量等于 dim(X)[MARGIN]的数组。如果函数 FUN 的结果是一个长度为 N 的向量，则结果是一个维数向量等于 c(N, dim(X)[MARGIN])的数组，注意这时不论是对哪一维计算，结果都放在了第一维。所以，比如我们要把 4×3 矩阵 a 的 3 列分别排序，只要用 apply(a, 2, sort)，这样对

每一列排序得到一个长度为 4 的向量，用第一维来引用，结果的维向量为 c(N, dim(a)[2])=c(4,3) ，保留了列维，恰好得到所需结果，运行如下例：

```
> a <- cbind(c(4,9,1), c(3,7,2))
> a
     [,1] [,2]
[1,]    4    3
[2,]    9    7
[3,]    1    2
> apply(a, 2, sort)
     [,1] [,2]
[1,]    1    2
[2,]    4    3
[3,]    9    7
>
```

但是，如果要对行排序，则 apply(a, 1, sort)把 a 的每一行 3 个元素排序后的结果用第一维来引用，结果的维向量为 c(N, dim(a)[1])=c(3, 4)，把原来的列变成了行，所以 t(apply(a,1,sort)) 才是对 a 的每一行排序的结果。如：

```
> apply(a, 1, sort)
     [,1] [,2] [,3]
[1,]    3    7    1
[2,]    4    9    2
> t(apply(a,1,sort))
     [,1] [,2]
[1,]    3    4
[2,]    7    9
[3,]    1    2
```

上面我们只用了矩阵（二维数组）作为例子讲解 apply 的用法。实际上，apply 可以用于任意维数的数组，函数 FUN 也可以是任意可以接收一个向量或数组作为第一自变量的函数。比如，设 x 是一个维数向量为 c(2,3,4,5)的数组，则 apply(x, c(1,3), sum)可以产生一个 2 行 4 列的矩阵，其每一元素是 x 中固定第 1 维和第 3 维下标取出子数组求和的结果。

因子和有序因子

统计中的变量有两个重要类别：区间变量和名义变量、有序变量。区间变量取连续的数值，可以进行求和、平均等运算。名义变量和有序变量取离散值，可以 用数值代表也可以是字符型值，其具体数值没有加减乘除的意义，不能用来计算而只能用来分类或者计数。名义变量比如性别、省份、职业，有序变量比如班级名 次。

因为离散变量有各种不同表示方法，在 S 中为统一起见使用因子(factor)来表示这种分类变量。还提供了有序因子(ordered factor)来表示有序变量。因子是一种特殊的字符型向量，其中每一个元素取一组离散值中的一个，而因子对象有一个特殊属性 levels 表示这组离散值（用字符串表示）。例如：

```
> x <- c("男", "女", "男", "男", "女")
> y <- factor(x)
> y
[1] 男 女 男 男 女
Levels: 男 女
```

函数 factor()用来把一个向量编码成为一个因子。其一般形式为：

factor(x, levels = sort(unique(x), na.last = TRUE), labels,

exclude = NA, ordered = FALSE)

可以自行指定各离散取值（水平 levels），不指定时由 x 的不同值来求得。labels 可以用来指定各水平的标签，不指定时用各离散取值的对应字 符串。exclude 参数用来指定要转换为缺失值（NA）的元素值集合。如果指定了 levels，则因子的第 i 个元素当它等于水平中第 j 个时元素值 取"j"，如果它的值没有出现在 levels 中则对应因子元素值取 NA。ordered 取真值时表示因子水平是有次序的（按编码次序）。

可以用 is.factor()检验对象是否因子，用 as.vector()把一个向量转换成一个因子。

因子的基本统计是频数统计，用函数 table()来计数。例如，

```
> sex = factor(c("男", "女", "男", "男", "女"))
> res.tab <- table(sex)
> res.tab
男 女
3  2
```

表示男性 3 人，女性 2 人。table()的结果是一个带元素名的向量，元素名为因子水平，元素值为该水平的出现频数。

从这个简单例子我们也可以看出 S 与 SAS 的一个区别：SAS 的结果一般只显示出来，保存只能存为数据集一种类型，不便于对中间结果进一步编程处理；而 S 的结果除了可以显示外本身都是 S 对象（如这里的向量结果），可以很方便地进一步处理。

可以用两个或多个因子进行交叉分类。比如，性别（sex）和职业（job）交叉分组可以用 table(sex, job)来统计每一交叉类的频数，结果为一个矩阵，矩阵带有行名和列名，分别为两个因子的各水平名。

因子可以用来作为另外的同长度变量的分类变量。比如，假设上面的 sex 是 5 个学生的性别，而

> h <- c(165, 170, 168, 172, 159)

是这 5 个学生的身高，则

> tapply(h, sex, mean)

可以求按性别分类的身高平均值。这样用一个等长的因子向量对一个数值向量分组的办法叫做不规则数组（ragged array）。后面我们还可以看到更多的因子的应用。

列表（list）

列表定义

列表是一种特别的对象集合，它的元素也由序号（下标）区分，但是各元素的类型可以是任意对象，不同元素不必是同一类型。元素本身允许是其它复杂数据类型，比如，列表的一个元素也允许是列表。例如：

```
> rec <- list(name="李明", age=30, scores=c(85, 76, 90))
> rec
$name
[1] "李明"

$age
[1] 30

$scores
[1] 85 76 90
```

列表元素总可以用“列表名[[下标]]”的格式引用。例如：

```
> rec[[2]]
[1] 30
> rec[[3]][2]
[1] 76
```

但是，列表不同于向量，我们每次只能引用一个元素，如 rec[[1:2]]的用法是不允许的。

注意：“列表名[下标]”或“列表名[下标范围]”的用法也是合法的，但其意义与用两重括号的记法完全不同，两重记号取出列表的一个元素，结果与该元素类型相同，如果使用一重括号，则结果是列表的一个子列表（结果类型仍为列表）。

在定义列表时如果指定了元素的名字（如 rec 中的 name，age，scores），则引用列表元素还可以用它的名字作为下标，格式为“列表名[["元素名"]]”，如：

```
> rec[["age"]]
[1] 30
另一种格式是“列表名$元素名”，如：
> rec$age
[1] 30
```

其中“元素名”可以简写到与其它元素名能够区分的最短程度，比如“rec$s”可以代表“rec$score”。这种写法方便了交互运行，编写程序时一般不用简写以免降低程序的可读性。

使用元素名的引用方法可以让我们不必记住某一个下标代表那一个元素，而直接用易记的元素名来引用元素。事实上，向量和矩阵也可以指定元素名、行名、列名。

定义列表使用 list()函数，每一个自变量变成列表的一个元素，自变量可以用“名字＝值”的方式给出，即给出列表元素名。自变量的值被复制到列表元素中，自变量如果是变量并不会与该列表元素建立关系（改变该列表元素不会改变自变量的值）。

修改列表

列表的元素可以修改，只要把元素引用赋值即可。如：

```
> rec$age <- 45
```

甚至

```
> rec$age <- list(19, 29, 31)
```

（可以任意修改一个列表元素）。如果被赋值的元素原来不存在，则列表延伸以包含该新元素。例如，rec 现在共有三个元素，我们定义一个新的命名元素，则列表长度变为 4，再定义第六号元素则列表长度变为 6：

```
> rec$sex <- "男"
> rec[[6]] <- 161
> rec
$name
[1] "李明"

$age
[1] 30

$scores
[1] 85 76 90

$sex
[1] "男"
```

```
[[5]]
NULL

[[6]]
[1] 161
```

第五号元素因为没有定义所有其值是“NULL”，这是空对象的记号。如果 rec 是一个向量，则其空元素为“NA”，这是缺失值的记号。从这里我们也可以体会“NULL”与“NA”的区别。

几个列表可以用连接函数 c()连接起来，结果仍为一个列表，其元素为各自变量的列表元素。如：

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

（注意在 S 中句点是名字的合法部分，一般没有特殊意义。）

几个返回列表的函数

列表的重要作用是把相关的若干数据保存在一个数据对象中，这样在编写函数时我们就可以返回这样一个包含多项输出的列表。因为函数的返回结果可以完整地存放在一个列表中，我们可以继续对得到的结果进行分析，这是 S 比 SAS 灵活的一个地方。下面给出几个返回列表的例子。

一、特征值和特征向量

函数 eigen(x)对对称矩阵 x 计算其特征值和特征向量，返回结果为一个列表，列表的两个成员（元素）为 values 和 vectors。例如：

```
> ev <- eigen((1:3) %o% (1:3))
> ev
$values
[1] 1.400000e+001 0.000000e+000-8.881784e-016

$vectors
          [,1]       [,2]       [,3]
[1,] 0.2672612  0.8944272 -0.3585686
[2,] 0.5345225 -0.4472136 -0.7171372
[3,] 0.8017837  0.0000000  0.5976143
```

可见三个特征值只有第一个不为零（由于数值计算精度所限，第三个特征值应为零但结果只是近似为零）。特征向量按矩阵存放，每一列为一个特征向量。

二、奇异值分解及行列式

函数 svd()进行奇异值分解 $X=UDV'$，其中 X 是任意 $n\times m$ 阵， U 为 $n\times n$ 正交阵， V 为 $m\times m$ 正交阵， D 为 $n\times m$ 对角阵（只有主对角线元素不为零）。svd(x) 返回有三个成员 d，u，v 的列表，d 为包含奇异值的向量（即 D 的主对角线元素），u，v 分别为上面的两个正交阵。

易见如果矩阵 x 是对称阵则 x 的行列式的绝对值等于奇异值的乘积，所以：

```
> absdetx <- prod(svd(x)$d)
```

或者我们可以为此定义一个函数：

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

三、最小二乘拟合与 QR 分解

函数 lsfit(x,y)返回最小二乘拟合的结果。最小二乘的模型为线性模型

$$y=X\beta+\varepsilon$$

lsfit(x,y)的第一个参数 x 为模型中的设计阵 X，第二个参数 y 为模型中的因变量 y（可以是一个向量也可以是一个矩阵），返回一个列表，成员 coefficients 为上面模型的 β（最小二乘系数），成员 residuals 为拟合残差，成员 intercept 用来指示是否有截距项，成员 qr 为设计阵 X 的 QR 分解，它本身也是一个列表。模型拟合缺省情况有截距项，可以用 intercept=FALSE 选项指定无截距项。关于最小二乘拟合还可参见 ls.diag() 函数（查看帮助）。

函数 qr(x)返回 x 的 QR 分解结果。矩阵 X 的 QR 分解为 $X=QR$， Q 为对角线元素都等于 1 的下三角阵， R 为上三角阵。函数结果为一个列表，成员 qr 为一个矩阵，其上三角部分（包括对角线）分解的 R，其下三角部分（不包括对角线）为分解的 Q。其它成员为一些辅助信息。

数据框（data.frame）

数据框是 S 中类似 SAS 数据集的一种数据结构。它通常是矩阵形式的数据，但矩阵各列可以是不同类型的。数据框每列是一个变量，每行是一个观测。

但是，数据框有更一般的定义。它是一种特殊的列表对象，有一个值为“data.frame”的 class 属性，各列表成员必须是向量（数值型、字符型、逻辑型）、因子、数值型矩阵、列表，或其它数据框。向量、因子成员为数据框提供一个变量，如果向量非数值型 则会被强制转换为因子，而矩阵、列表、数据框这样的成员为新数据框提供了和其列数、成员数、变量数相同个数的变量。作为数据框变量的向量、因子或矩阵必须 具有相同的长度（行数）。

尽管如此，我们一般还是可以把数据框看作是一种推广了的矩阵，它可以用矩阵形式显示，可以用对矩阵的下标引用方法来引用其元素或子集。

数据框生成

数据框可以用 data.frame()函数生成，其用法与 list()函数相同，各自变量变成数据框的成分，自变量可以命名，成为变量名。例如：

```
> d <- data.frame(name=c("李明", "张聪", "王建"), age=c(30, 35, 28),
+       height=c(180, 162, 175))
> d
  name age height
1 李明  30    180
2 张聪  35    162
3 王建  28    175
```

如果一个列表的各个成分满足数据框成分的要求，它可以用 as.data.frame()函数强制转换为数据框。比如，上面的 d 如果先用 list()函数定义成了一个列表，就可以强制为一个数据框。

一个矩阵可以用 data.frame()转换为一个数据框，如果它原来有列名则其列名被作为数据框的变量名，否则系统自动为矩阵的各列起一个变量名（如 X1，X2）。

数据框引用

引用数据框元素的方法与引用矩阵元素的方法相同，可以使用下标或下标向量，也可以使用名字或名字向量。如 d[1:2, 2:3]。数据框的各变量也可以用按列表引用（即用双括号[[]] 或$符号引用）。

数据框的变量名由属性 names 定义，此属性一定是非空的。数据框的各行也可以定义名字，可以用 rownames 属性定义。如：

```
> names(d)
```

```
[1] "name"   "age"    "height"
> rownames(d)
[1] "1" "2" "3"
```

attach()函数

数据框的主要用途是保存统计建模需要的数据。S 的统计建模功能都需要以数据框为输入数据。我们也可以把数据框当成一种矩阵来处理。

在使用数据框的变量时可以用“数据框名$变量名”的记法。但是，这样使用较麻烦，S 提供了 attach()函数可以把数据框“连接”入当前的名字空间。例如，

> attach(d)

> r <- height / age

后一语句将在当前工作空间建立一个新变量 r，它不会自动进入数据框 d，要把新变量赋值到数据框中，可以用

> d$r <- height / age

这样的格式。

为了取消连接，只要调用 detach()（无参数即可）。

注意：S 和 R 中名字空间的管理是比较独特的。它在运行时保持一个变量搜索路径表，在读取某个变量时到这个变量搜索路径表中由前向后查找，找到最前的一个；在赋值时总是在位置 1 赋值（除非特别指定在其它位置赋值）。attach()的缺省位置是在变量搜索路径表的位置 2，detach()缺省也是去掉位置 2。所以，S 编程的一个常见问题是当你误用了一个自己并没有赋值的变量时有可能不出错，因为这个变量已在搜索路径中某个位置有定义，这样不利于程序的调试，需要留心这样的问题。

除了可以连接数据框，也可以连接列表。

输入输出

输出

在 S 交互运行时要显示某一个对象的值只要键入其名字即可，如：

```
> x <- 1:10
> x
[1]  1  2  3  4  5  6  7  8  9 10
```

这实际上是调用了 print()函数，即 print(x)。在非交互运行（程序）中应使用 print()来输出。print()函数可以带一个 digits=参数指定每个数输出的有效数字位数，可以带一个 quote= 参数指定字符串输出时是否带两边的撇号，可以带一个 print.gap=参数指定矩阵或数组输出时列之间的间距。

print()函数是一个通用函数，即它对不同的自变量有不同的反应。对各种特殊对象如数组、模型结果等都可以规定 print 的输出格式。

cat()函数也用来输出，但它可以把多个参数连接起来再输出（具有 paste()的功能）。例如：

> cat("i = ", i, "\n")

注意使用 cat()时要自己加上换行符"\n"。它把各项转换成字符串，中间隔以空格连接起来，然后显示。如果要使用自定义的分隔符，可以用 sep=参数，例如：

```
> cat(c("AB", "C"), c("E", "F"), "\n", sep="")
ABCDEF
```

cat()还可以指定一个参数 file=给一个文件名，可以把结果写到指定的文件中，如：

```
> cat("i = ", 1, "\n", file="c:/work/result.txt")
```

如果指定的文件已经存在则原来内容被覆盖。加上一个 append=TRUE 参数可以不覆盖原文件而是在文件末尾附加，这很适用于运行中的结果记录。

cat()函数和 print()都不具有很强的自定义格式功能，为此可以使用 cat()与 format()函数配合实现。format()函数为一个数值向量找到一种共同的显示格式然后把向量转换为字符型。例如：

```
> format(c(1, 100, 10000))
[1] "    1" "  100" "10000"
```

S-PLUS 中的 format()函数功能较强，具有较多的控制参数，请参见帮助。R 中目前 format() 函数功能仍较弱，但 R 有一个 formatC 函数可以提供类似 C 语言的 printf 格式功能。formatC 对输入向量的每一个元素单独进行格式转换而不生成统一格式，例如：

```
> formatC(c(1, 10000))
```

```
[1] "1"      "1e+004"
```

在 formatC()函数中可以用 format=参数指定 C 格式类型，如"d"（整数），"f"'（定点实数），"e"（科学记数法），"E", "g"（选择位数较少的输出格式）,"G", "fg"（定点实数但用 digits 指定有效位数），"s"（字符串）。可以用 width 指定输出宽度，用 digits 指定有效位数（格式为 e,E,g,G,fg 时）或小数点后位数（格式为 f）时。可以用 flag 参数指定一个输出选项字符串，字符串中有"-"表示输出左对齐，有"0"表示左空白用 0 填充，有"+"表示要输出正负号，等等。例 如，我们有一个矩阵 da 中保存了三个日期的年、月、日：

```
> da
     [,1] [,2] [,3]
[1,]   99    1    3
[2,]   96   11    9
[3,]   65    5   18
```

为了输出这三个日期，可以用 apply 函数指定对每一行作用一个输出函数，此输出函数利用 cat()和 formatC 来控制：

```
> apply(da, 1, function(r)
+  cat(formatC(r[1], format='d', width=2, flag='0'), '-',
+  formatC(r[2], format='d', width=2, flag='0'), '-',
+  formatC(r[3], format='d', width=2, flag='0'), '\n', sep=''))
99-01-03
96-11-09
65-05-18
NULL
```

这里我们知道 apply 函数第一个参数指定了一个矩阵，第二个参数说明对行操作还是对列操作，第三个参数是一个函数，这里我们使用了直接定义一个函 数作为参数的办法。输出结果中多了一个 NULL 函数，这是因为我们在交互运行，apply 的结果作为一个表达式的值（NULL ）会被显示出来。为避免显示，可以把结果赋给一个临时变量名，或者把整个表达式作为 invisible() 函数的参数，这时不显示表达式值。

S 的输出缺省显示在交互窗口。可以用 sink()函数指定一个文件以把后续的输出转向到这个文件，并可用 append 参数指定是否要在文件末尾附加：

```
> sink("c:/work/result.txt", append=TRUE)
> ls()
> d
> sink()
```

调用无参数的 sink()把输出恢复到交互窗口。

输入

为了从外部文件读入一个数值型向量，S 提供了 scan()函数。如果指定了 file 参数（也是第一参数），则从指定文件读入，缺省情况下读入一个数值向量，文件中各数据以空白分隔，读到文件尾为止。例如：

```
> cat(1:12, '\n', file='c:/work/result.txt')

> x <- scan('c:/work/result.txt')
```

如果文件中是一个用空白分隔的矩阵（或数组），我们可以先用 scan()把它读入到一个向量然后用 matrix()函数（或 array()函数）转换。如：

```
> y <- matrix(scan('c:/work/result.txt'), ncol=3, byrow=T)
```

实际上，scan()也能够读入一个多列的表格，只要用 what 参数指定一个列表，则列表每项的类型为需要读取的类型。用 skip 参数可以跳过文件的开始若干行不读。用 sep 参数可以指定数据间的分隔符。详见帮助。

scan()不指定读取文件名时是交互读入，读入时用一个空行结束。

如果要读取一个数据框，S 提供了一个 read.table()函数。它只要给出一个文件名，就可以把文件中用空白分隔的表格数据每行读入为数据框的一行。比如，文件 c:\work\d.txt 中内容如下：

```
Zhou  15  3
"Li Ming" 9  李明
Zhang 10.2 Wang
```

用 read.table 读入：

```
> x <- read.table('c:/work/d.txt', as.is=T)
> x
       V1   V2   V3
1 Zhou    15   3
2 Li Ming 9    李明
3 Zhang   10.2 Wang
```

读入结果为数据框。函数可以自动识别表列是数值型还是字符型，并在缺省情况下把字符型数据转换为因子（加上 as.is=T 可以保留字符型不转换）。函数自动为数据框变量指定“V1 ”、“V2”这样的变量名，指定“1”、“2”这样的行名。可以用 col.names 参数指定一个字符型向量作为数据框的变量名，用 row.names 参数指定一个字符型向量作为数据框的行名。

read.table()可以读入带有表头的文件，只要加上 header=TRUE 参数即可。可以用 sep 参数指定表行各项的分隔符。例如，为了读入如下带有表头的逗号分隔文件 c:\work\d.csv：

```
Name,score, cn
Zhou,15,3
Li Ming, 9,  李明
Zhang, 10.2, Wang
```

使用如下语句：

```
> x <- read.table('c:/ldf/tmp.txt', header=T, sep=',')
> x
     Name score     cn
1    Zhou  15.0      3
2 Li Ming   9.0   李明
3   Zhang  10.2   Wang
```

其它一些用法见帮助。

程序控制结构

S 是一个表达式语言，其任何一个语句都可以看成是一个表达式。表达式之间以分号分隔或用换行分隔。表达式可以续行，只要前一行不是完整表达式（比如末尾是加减乘除等运算符，或有未配对的括号）则下一行为上一行的继续。

若干个表达式可以放在一起组成一个复合表达式，作为一个表达式使用。组合用大括号表示，如：

```
> {
>  x <- 15
>  x
> }
```

S 语言也提供了其它高级程序语言共有的分支、循环等程序控制结构。

分支结构

分支结构包括 if 结构：

```
if (条件) 表达式 1
```

或

```
if (条件) 表达式 1 else 表达式 2
```

其中的“条件”为一个标量的真或假值，表达式可以是用大括号包围的复合表达式。有 else 子句时一般写成：

```
if(条件) {
        表达式组……
} else {
        表达式组……
}
```

这样的写法可以使 else 不至于脱离前面的 if。例如，如果变量 lambda 为缺失值就给它赋一个缺省值，可用：

```
if(is.na(lambda)) lambda <- 0.5;
```

又比如要计算向量 x 的重对数，这只有在元素都为正且对数都为正时才能做到，因此需要先检查：

```
if(all(x>0) && all(log(x))>0) {
  y <- log(log(x));
  print(cbind(x,y));
  }
else{
  cat('Unable to comply\n');
  }
```

注意“&&”表示“与”，它是一个短路运算符，即第一个条件为假时就不计算第二个条件，如果不这样此例中计算对数就可以有无效值。在条件中也可以用“||”（两个连续的竖线符号）表示“或”，它也是短路运算符，当第一个条件为真时就不再计算第二个条件。

在用 S 编程序时一定要时刻牢记 S 是一个向量语言，几乎所有操作都是对向量进行的。而 S 中的 if 语句却是一个少见的例外，它的判断条件是标量的真值或假值。比如，我们要定义一个分段函数 f(x)，当 x 为正时返回 1，否则返回 0，马上可以想到用 if 语句实现如下：

if(x>0) 1 else 0

当 x 是标量时这个定义是有效的，但是当自变量 x 是一个向量时，比较的结果也是一个向量，这时条件无法使用。所以，这个分段函数应该这样编程：

```
y <- numeric(length(x))
y[x>0] <- 1
y[x<=0] <- 0
y
```

有多个 if 语句时 else 与最近的一个配对。可以使用 if ... else if ... else if ... else ...的多重判断结构表示多分支。多分支也可以使用 switch()函数。

循环结构

循环结构中常用的是 for 循环，是对一个向量或列表的逐次处理，格式为“for(*name* in *values*) 表达式”，如：

```
for(i in seq(along=x){
  cat('x(', i, ') = ', x[i], '\n', sep='');
  s <- s+x[i];
}
```

这个例子我们需要使用下标的值，所以用 seq(along=x)生成了 x 的下标向量。如果不需要下标的值，可以直接如此使用：

```
for(xi in x){
  cat(xi, '\n')
  s <- s + xi
}
```

当然，如果只是要求各元素的和，只要调用 sum(x)即可。从这里我们也可以看出，显式的循环经常是可以避免的，利用函数对每个元素计算值、使用 sum 等统计函数及 apply、lapply 、sapply、tapply 等函数往往可以代替循环。因为循环在 S 中是很慢的（S-PLUS 和 R 都是解释语言），所以应尽可能避免使用显式循环。

我们再举一个例子。比如，我们要计算同生日的概率。假设一共有 365 个生日（只考虑月、日），而且各生日的概率是相等的（这里忽略了闰年的情况以及 可能存在的出生日期分布的不均匀）。设一个班有 n 个人，当 n 大于等于 365 时至少两个人生日相同是必然时间。当 n 小于 365 时，我们可以计算 P{至少有两 人同生日}= 1 - P{n 个人生日彼此不同}，这时，n 个人的生日可取值数为 365^n，而 n 个人彼此不同的可能数为 365 中取 n 个的排列数，彼此不同的概率为 $365\times364\times\cdots\times(365-(n-1))=365!/(365-n)!$。因此，为了计算 n＝1，2，...，364 的情况下的同生日概率，可以用如下循环实现：

```
> x <- numeric(364)
> for(i in 1:364){
+   x[i] <- 1
+   for(j in 0:(i-1))
+     x[i] <- x[i] * (365-i)/365
+   x[i] <- 1 - x[i]
+ }
```

这段程序运行了 36 秒。我们可以尽量用向量运算来实现，速度要快得多：

```
> x <- numeric(364)
> for(n in 1:364){
+   x[n] <- 1 - prod((365:(365-n+1))/365)
+ }
```

这段程序只用了 1 秒。注意不能直接去计算 365!，这会超出数值表示范围。

另外要注意使用 for(i in 1:n)格式的计数循环时要避免一个常见错误，即当 n 为零或负数时 1:n 是一个从大到小的循环，而我们经常需要的是当 n 为零或负数时就不进入循环。为达到这一点，可以在循环外层判断循环结束值是否小于开始值。

while 循环是在开始处判断循环条件的当型循环，如：

```
while(b-a>eps){
  c <- (a+b)/2;
  if(f(c)>0) b <- c
  else a <- c
  }
```

是一段二分法解方程的程序。

还可以使用

```
repeat 表达式
```

循环，在循环体内用 break 跳出。

在一个循环体内用 next 表达式可以进入下一轮循环。

分支和循环结构主要用于定义函数。

S 程序设计

对于复杂一些的计算问题我们应该编写成函数。这样做的好处是编写一次可以重复使用，并且可以很容易地修改，另外的好处是函数内的变量名是局部的，运行函数不会使函数内的局部变量被保存到当前的工作空间，可以避免在交互状态下直接赋值定义很多变量使得工作空间杂乱无章。

工作空间管理

前面我们已经提到，S 在运行时保持一个变量搜索路径表，要读取某变量时依次在此路径表中查找，返回找到的第一个；给变量赋值时在搜索路径的第一个位 置赋

值。但是，在函数内部，搜索路径表第一个位置是局部变量名空间，所以变量赋值是局部赋值，被赋值的变量只在函数运行期间有效。

用 ls()函数可以查看当前工作空间保存的变量和函数，用 rm()函数可以剔除不想要的对象。如：

```
> ls()
 [1] "A"      "Ai"     "b"      "cl"     "cl.f"   "fit1"   "g1"     "marks"
"ns"
[10] "p1"     "rec"    "tmp.x" "x"      "x1"     "x2"     "x3"     "y"
> rm(x, x1, x2, x3)
> ls()
 [1] "A"      "Ai"     "b"      "cl"     "cl.f"   "fit1"   "g1"     "marks"
"ns"
[10] "p1"     "rec"    "tmp.x" "y"
```

ls()可以指定一个 pattern 参数，此参数定义一个匹配模式，只返回符合模式的对象名。模式格式是 UNIX 中 grep 的格式。比如，ls(pattern="tmp[.]")可以返回所有以“tmp.” 开头的对象名。

rm()可以指定一个名为 list 的参数给出要删除的对象名，所以 rm(list=ls(pattern="tmp[.]")) 可以删除所有以“tmp.”开头的对象名。

函数定义

S 中函数定义的一般格式为“函数名 <- function(参数表) 表达式”。定义函数可以在命令行进行，例如：

```
> hello <- function(){
+   cat("Hello, world\n")
+   cat("\n")
+ }
> hello
function ()
{
            cat("Hello, world\n")
            cat("\n")
}
> hello()
Hello, world
```

函数体为一个复合表达式，各表达式的之间用换行或分号分开。不带括号调用函数显示函数定义，而不是调用函数。

在命令行输入函数程序很不方便修改，所以我们一般是打开一个其他的编辑程序（如 Windows 的记事本），输入以上函数定义，保存文件，比如保存到了 C:\R\hello.s，我们就可以用

> source("c:\\R\\hello.s")

运行文件中的程序。实际上，用 source()运行的程序不限于函数定义，任何 S 程序都可以用这种方式编好再运行，效果与在命令行直接输入是一样的。

对于一个已有定义的函数，可以用 fix()函数来修改，如：

```
> fix(hello)
```

将打开一个编辑窗口显示函数的定义，修改后关闭窗口函数就被修改了。fix()调用的编辑程序缺省为记事本，可以用“options(editor="编辑程序名")”来指定自己喜欢的编辑程序。

函数可以带参数，可以返回值，例如：

```
larger <- function(x, y){
  y.is.bigger <- (y>x);
  x[y.is.bigger] <- y[y.is.bigger]
  x
}
```

这个函数输入两个向量（相同长度）x 和 y，然后把 x 中比 y 对应元素小的元素替换为 y 中对应元素，返回 y 的值。S 返回值为函数体的最后一个表达式的值，不需要使用 return()函数。不过，也可以使用“return(*对象*)”函数从函数体返回调用者。

参数（自变量）

函数可以带虚参数（形式自变量）。S 函数调用方式很灵活，例如，如下函数：

```
fsub <- function(x, y) x-y
```

有两个虚参数 x 和 y，我们用它计算 100－45，可以调用 fsub(100,45)，或 fsub(x=100,y=45) ，或 fsub(y=45, x=100)，或 fsub(y=45, 100)。即调用时实参与虚参可以按次序结合，也可以直接指定虚参名结合。实参先与指定了名字的虚参结合，没有指定名字的按次序与剩下的虚参结合。

函数在调用时可以不给出所有的实参，这需要在定义时为虚参指定缺省值。例如上面的函数改为：

fsub <- function(x, y=0) x-y

则调用时除了可以用以上的方式调用外还可以用 fsub(100)，fsub(x=100)等方式调用，只给出没有缺省值的实参。

即使没有给虚参指定缺省值也可以在调用时省略某个虚参，然后函数体内可以用 missing() 函数判断此虚参是否有对应实参，如：

```
trans <- function(x, scale){
  if(!missing(scale)) x <- scale*x
  ……
}
```

此函数当给了 scale 的值时对自变量 x 乘以此值，否则保持原值。这种用法在其它语言中是极其少见的，S 可以实现这一点是因为 S 的函数调用在用到参数的值时才去计算这个参数的值（称为“懒惰求值”），所以可以在调用时缺少某些参数而不被拒绝。

S 函数还可以有一个特殊的“...”虚参，表示所有不能匹配的实参，调用时如果有需要与其它虚参结合的实参必须用“虚参名＝”的格式引入。例如：

```
> f <- function(...){
+   for(x in list(...))
+   cat(min(x), '\n')
+ }
> f(c(5,1,2), c(9, 4, 7))
1
4
```

作用域

函数的虚参完全是按值传递的，改变虚参的值不能改变对应实参的值。例如：

```
> x <- list(1, "abc")
> x
[[1]]
[1] 1

[[2]]
[1] "abc"

> f <- function(x) x[[2]] <- "!!"
> f(x)
> x
[[1]]
[1] 1

[[2]]
[1] "abc"
```

函数体内的变量也是局部的，对函数体内的变量赋值当函数结束运行后变量值就删除了，不影响原来同名变量的值。例如：

```
> x <- 2
> f <- function(){
+   print(x)
+   x <- 20
+ }
> f()
[1] 2
> x
[1] 2
```

这个例子中原来有一个变量 x 值为 2，函数中为变量 x 赋值 20，但函数运行完后原来的 x 值并未变化。但是也要注意，函数中的显示函数调用时局部变量 x 还没有赋值，显示的是全局变量 x 的值。这是 S 编程比较容易出问题的地方：你用到了一个局部变量的值，你没有意识到这个局部变量还没有赋值，而程序却没有出错，因为这个变量已有全局定义。

程序调试

S-PLUS 和 R 目前还不象其它主流程序设计语言那样具有单步跟踪、设置断点、观察表达式等强劲的调试功能。调试复杂的 S 程序，可以用一些通用的程序调试方法，另外 S 也提供了一些调试用函数。

对任何程序语言，最基本的调试手段当然是在需要的地方显示变量的值。可以用 print() 或 cat()显示。例如，我们为了调试前面定义的 larger()函数，可以显示两个自变量的值及中间变量的值：

```
larger <- function(x, y){
  cat('x = ', x, '\n')
  cat('y = ', y, '\n')
  y.is.bigger <- (y>x);
  cat('y.is.bigger = ', y.is.bigger, '\n')
  x[y.is.bigger] <- y[y.is.bigger]
  x
}
```

S 提供了一个 browser()函数，当调用时程序暂停，用户可以查看变量或表达式的值，还可以修改变量。例如：

```
larger <- function(x, y){
  y.is.bigger <- (y>x);
  browser()
  x[y.is.bigger] <- y[y.is.bigger]
  x
```

```
}
```
我们运行此程序：

```
> larger(c(1,5), c(2, 4, 9))
Warning in y > x : longer object length
        is not a multiple of shorter object length
Called from: larger(c(1, 5), c(2, 4, 9))
Browse[1]> y
[1] 2 4 9
Browse[1]> x
[1] 1 5
Browse[1]> y>x
Warning in y > x : longer object length
        is not a multiple of shorter object length
[1]  TRUE FALSE  TRUE
Browse[1]> c
Error: subscript (3) out of bounds, should be at most 2
```

退出 R 的 browser()菜单可用 c（在 S 中用 return()）。在 R 的 browser()状态下用 n 命令可以进入单步执行状态，用 n 或者回车可以继续，用 c 可以退出。

R 提供了一个 debug()函数，debug(f)可以打开对函数 f()的调试，执行到函数 f 时自动进入单步执行的 browser()菜单。用 undebug(f)关闭调试。

程序设计举例

设计 S 程序是很容易的，在初学时我们只要使用我们从一般程序设计中学来的知识并充分利用 S 中现成的各种算法及绘图函数就可以了。但是，如果要用 S 编制计算量较大的程序，或者程序需要发表，就需要注意一些 S 程序设计的技巧。

用 S 语言开发算法，最重要的一点是要记住 S 是一个向量语言，计算应该尽量通过向量、矩阵运算来进行，或者使用 S 提供的现成的函数，避免使用显式循环。显式循环会大大降低 S 的运算速度，因为 S 是解释执行的。

比如，考虑核回归问题。核回归是非参数回归的一种，假设变量 Y 与变量 X 之间的关系为：

$$Y = f(X) + \varepsilon$$

其中函数 f 未知。观测到 X 和 Y 的一组样本 X_i, Y_i， i=1,..., n 后，对 f 的一种估计为：

$$f(x)=\sum_{i=1}^{n}K(\frac{x-X_i}{h})Y_i \Big/ \sum_{i=1}^{n}K(\frac{x-X_i}{h})$$

其中 K 叫做核函数，一般是一个非负的偶函数，原点处的函数值最大，在两侧迅速趋于零。例如正态密度函数，或所谓双三次函数核：

$$K(x)=\begin{cases}(1-|x|^3)^3 & |x|\le 1\\ 0 & \text{其它}\end{cases}$$

函数 density()可以进行核回归估计，这里作为举例我们对这个算法进行编程。先来编制双三次核函数的程序：

```
kernel.dcube <- function(u){
  y <- numeric(length(u))
  y[abs(u)<1] <- (1 - abs(u[abs(u)<1])^3)^3
  y[abs(u)>=1] <- 0
  y
}
```

注意上面分段函数不用 if 语句而是采用逻辑向量作下标的办法，这样定义出的函数允许以向量和数组作自变量。

假设我们要画出核估计曲线，一般我们取一个范围与各 X_i 范围相同的等间距向量 x 然后计算估计出的 $f(x)$的值。设观测数据自变量保存在向量 X 中，因变量保存在向量 Y 中，则按我们一般的程序设计思路，可以写成下面的 S 程序：

```
kernel.smooth1 <- function(X, Y, kernel=kernel.dcube, h=1, m=100,
plot.it=T){
  x <- seq(min(X), max(X), length=m)
  fx <- numeric(m)
  for(j in 1:m){
    # 计算第 j 个等间距点的回归函数估计值
    fx[j] <- sum(kernel((x[j]-X)/h)*Y) / sum(kernel((x[j]-X)/h))
  }
  if(plot.it){
    plot(X, Y, type="p")
    lines(x, fx)
  }
  cbind(x=x, fx=fx)
}
```

注意上面程序中用了 sum 函数来避免一重对 *i* 的循环。但是，上面的写法中仍有一重对 j 的循环，使得程序运行较慢。如何改写程序把这个循环也取消呢？办法是把计算看成是矩阵运算。首先，如果 x 是一个标量，则 f(x)可以写成：

$$f(x) = {K \cdot Y}/{K \cdot \mathbf{1}}$$

其中 $K = K\left(\frac{x - X}{h}\right)$ 是一个与 *X* 长度相等（即长度为 *n*）的行向量， $\mathbf{1}$ 是一列 1。现在，x 实际是一个长度为 *m* 的向量，对 x 的每一个元素可以计算一个长度为 *n* 的行向量 $K\left(\frac{x[j] - X}{h}\right)$，把这些行向量上下合并为一个矩阵 *K*，则 *KY* 为长度为 *m* 的向量，每一个元素是对应于一个 x[j]的分子。合并的矩阵可以用 S 的 outer() 函数来计算：

```
K <- outer(x, X, function(u, v) kernel((u-v)/h))

# outer()的第一个自变量对应于结果矩阵的行，第二个自变量对应于列
```

分母为了算每一行的和只要对 *K* 右乘 $\mathbf{1}$，于是结果的估计值向量为：

```
fx <- (K %*% Y) / (K %*% matrix(1,ncol=1,nrow=length(Y))
```

这样修改 kernel.smooth1 可以得到更精简的函数 kernel.smooth2。这种方法在 R 中可以实现，但在 Splus 中运行却有问题，因为 Splus 不允许函数内部再定义函数。

核回归中窗宽 *h* 的选择是比较难的，我们写的核回归函数应该允许用户输入 *h* 为一个向量，对向量中每一个窗口计算一条拟合曲线并画在图中，结果作为函数返回值的一列。读者可以作为练习实现这个函数，并模拟 X 和 Y 的观测然后画核估计曲线， *f*(*x*)用 sin(x)。对 *h* 我们可能必须用循环来处理了。

图形

常用图形

S-PLUS 有很强的图形功能，它可以用简单的函数调用迅速作出数据的各种图形，当你熟悉了 S 图形的技术之后也可以指定许多图形选项按自己的要求定制图形。

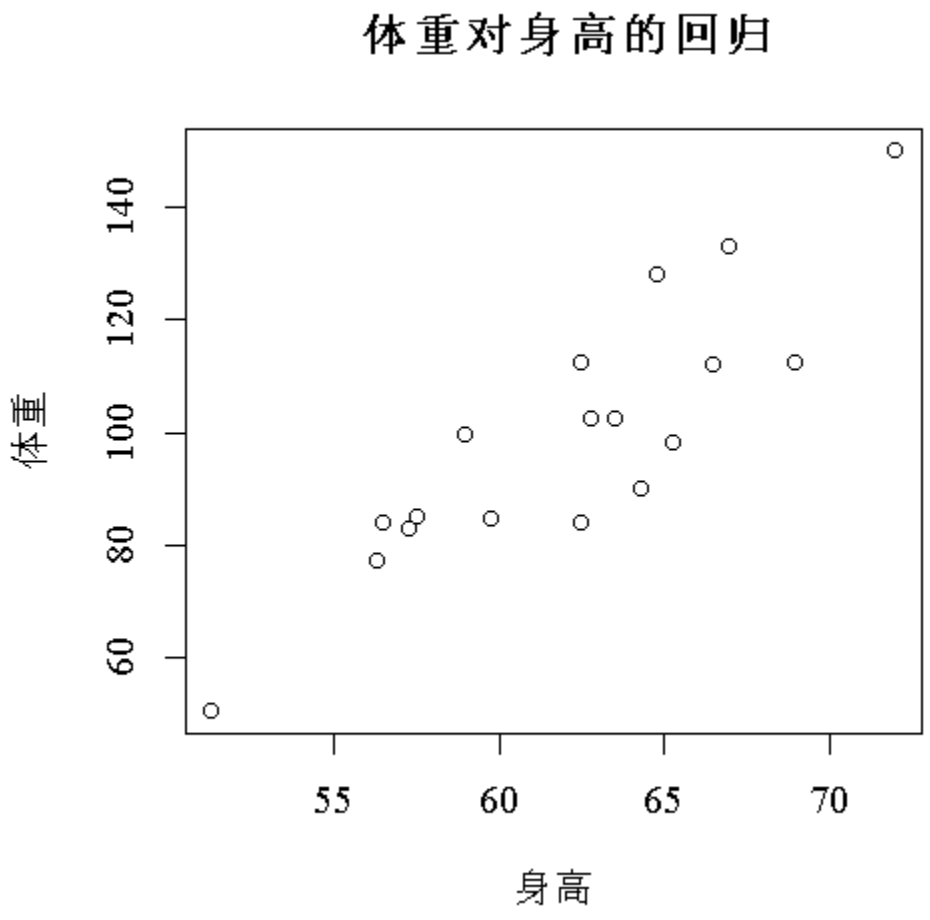
它的另一个特色是同一个绘图函数对不同的数据对象可以作出不同的图形。例如，用 7.1.2 读入的 cl 数据框：

```
> plot(cl)
> plot(cl[,1])
```

第一个 plot()调用绘制 cl 中三个列的散点图矩阵，第二个 plot()调用绘制身高的散点图（纵轴为身高值，横轴为下标）。

最常用的绘图函数为 plot()，用 plot()作两个变量 x 与 y 的散点图，使用如下例的方法：

```
> attach(cl)
> plot(Height, Weight, main="体重对身高的回归",
+ xlab="身高", ylab="体重")
```



上例在 R 中运行成功（见图 2），在 S-PLUS 中请将汉字字符串改为英文。上例也演示了 S 中如何输入较长的语句：只要语句明显地未完成（比如，缺右括号），系统将给出一个加号作为续行提示。如果输入“x <- 1+2”时要拆行，可以在赋值号后拆，可以在加号与 2 之间拆，但是如果在 x 后拆则只能显示 x 的当前值，如果在 1 与加号之间拆只能把 1 赋给 x。

为了绘制连线图，只要在 plot()函数中加 type="l"选项，如：

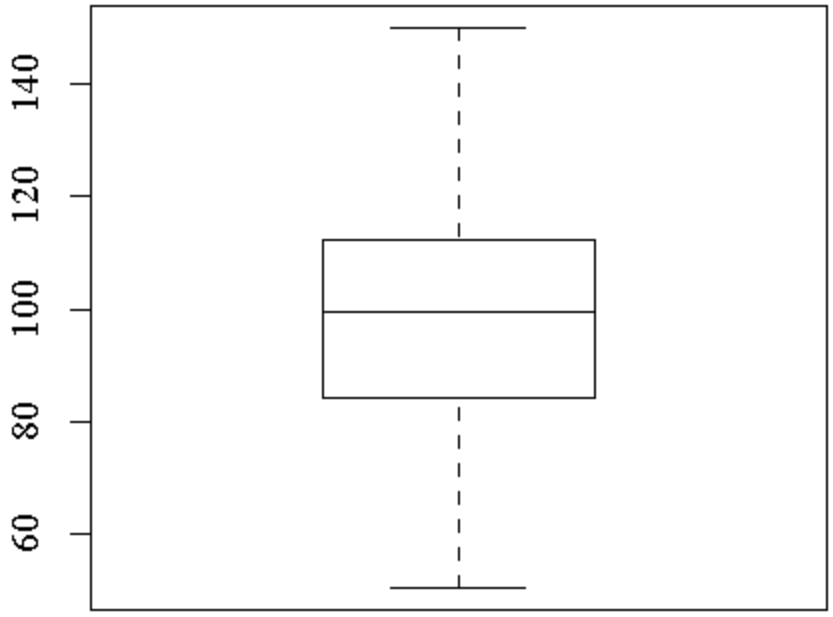
```
> plot((1:50)/50, log((1:50)/50), type="l")
```

可以绘制变量的茎叶图，如：

```
> stem(Weight)

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

   4 | 1
   6 | 7
   8 | 3445508
  10 | 0332233
  12 | 83
  14 | 0
```

绘制一个变量的盒形图，如：

```
> boxplot(Weight)
```

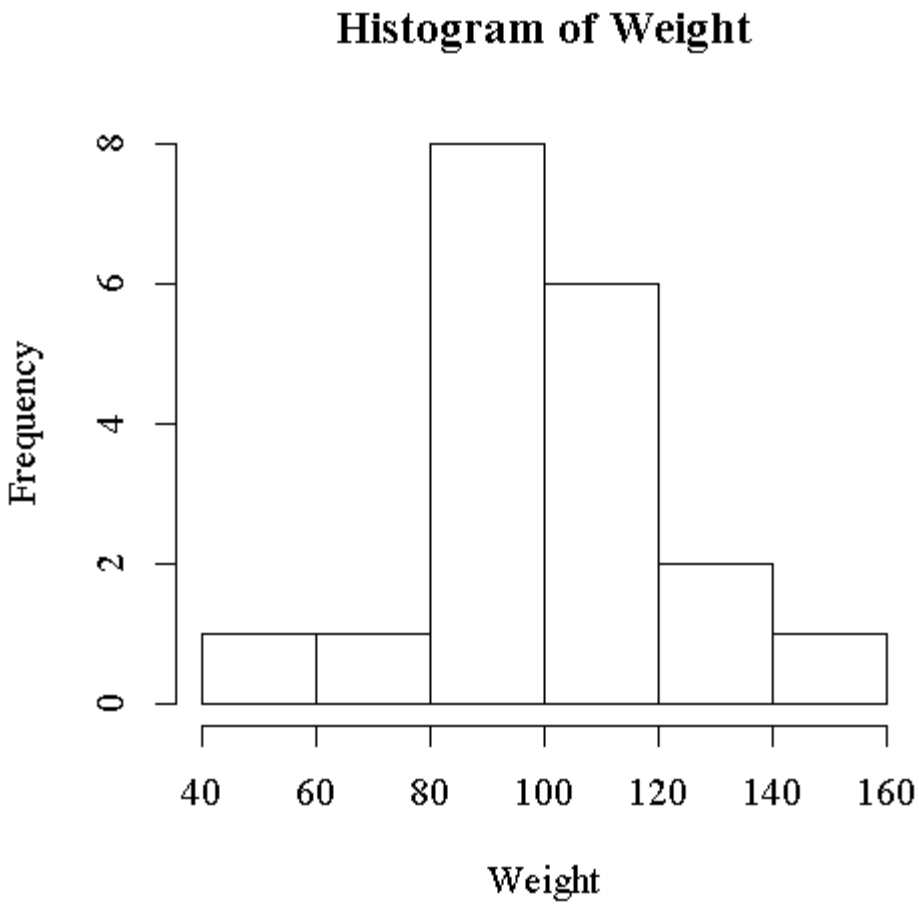


结果见图 3。可以绘制几个变量并排的盒形图，比如先计算用上面的回归拟合结果存入 p1 ，然后绘制并排盒形图：

```
> p1 <- predict(fit1, cl)[,"predictor"]
> boxplot(Weight, p1)
```

用 hist()函数可以绘制直方图。例如：

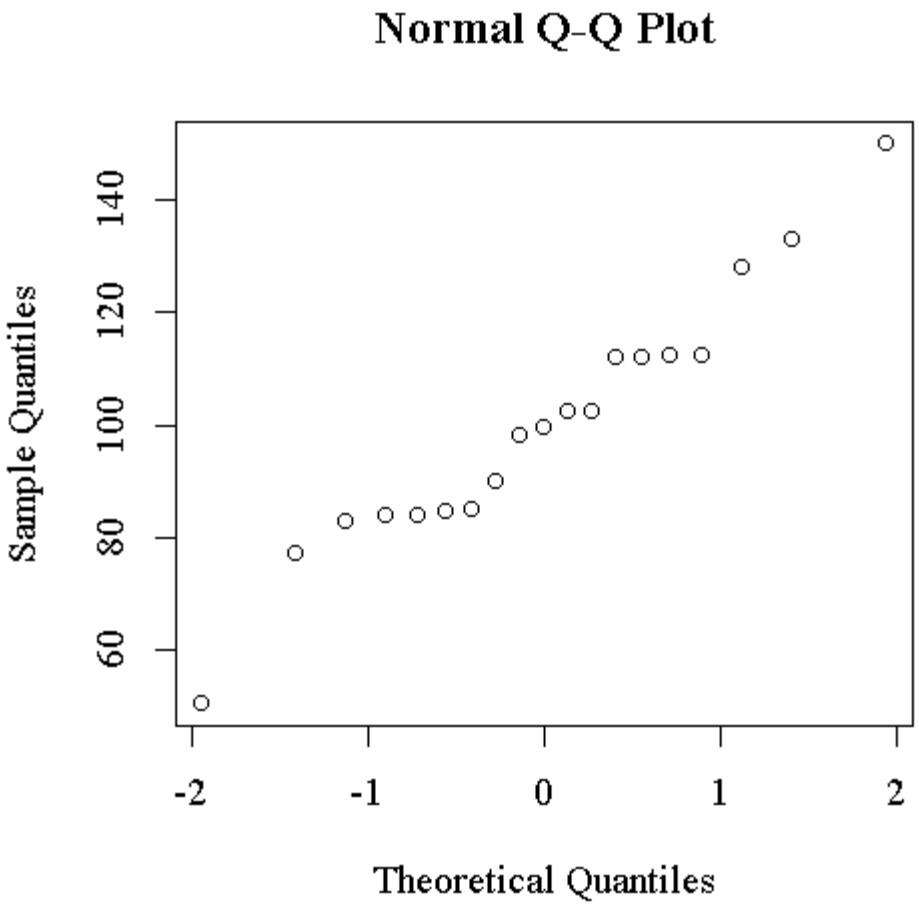
```
> hist(Weight)
```



得图 4。

用 qqnorm()函数绘制正态概率图，如：

```
> qqnorm(Weight)
```



得图 5。

高级图形函数

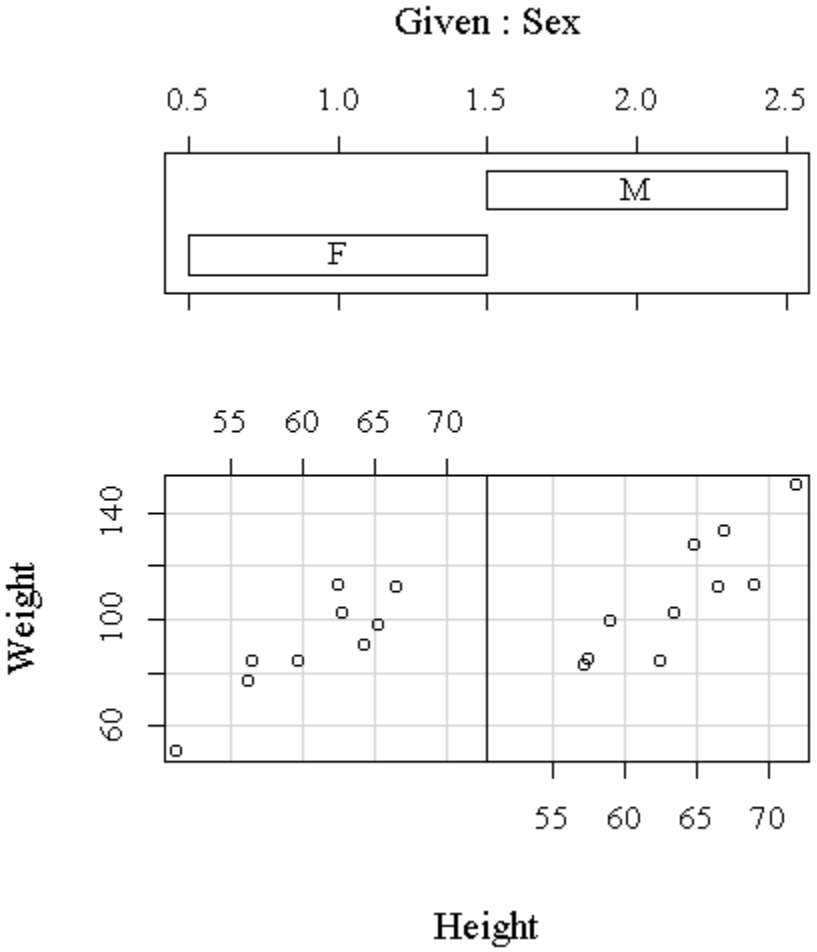
S 的图形函数分为两类：高级图形函数——直接绘制图形并可自动生成坐标轴等附属图形元素；低级图形函数——可以修改已有的图形或者为绘图规定一些选择项。高级图形函数总是开始一个新图。下面我们介绍常用的高级图形函数，以及用来修饰这些高级图形函数的常用可选参数。

最常用的是 plot()函数。比如，plot(x,y)（其中 x，y 是向量）对两个变量画散点图。用 plot(z) （其中 z 是一个定义了 x 变量和 y 变量的列表，或者一个两列的矩阵）也可以达到同样目的。如果 x 是一个时间序列对象（时间序列对象用 ts()函数生 成），plot(x)绘制时间序列曲线图。如果 x 是一个普通向量，则绘制 x 的值对其下标的散点图。如果 x 是复数向量则绘制虚部对实部的散点图。如果 f 是一 个因子，则 plot(f)绘制 f 的条形图（每个因子水平的个数）。如果 f 是因子，y 是同长度的数值向量，则 plot(f,y)对 f 的每一因子水平绘制 y 中 相应数值的盒形图。如果 d 是一个数据框，则 plot(d)对 d 的每两个变量之间作图（散点图等）。

如果 X 是一个数值型矩阵或数据框，用 pairs(X)可以绘制每两列之间的散点图矩阵。这在变量个数不太多时可以同时看到多个变量的两两关系，变量太多时则难以绘制。

协同图（coplot）是一种多变量的探索性分析图形。其形式为 coplot(y ~ x | z)，其中 x 和 y 是数值型向量，z 是同长度的因子。对 z 的每一水平，绘制相应组的 x 和 y 的散点图。如:

```
> attach(cl)
> coplot(Weight ~ Height | Sex)
```



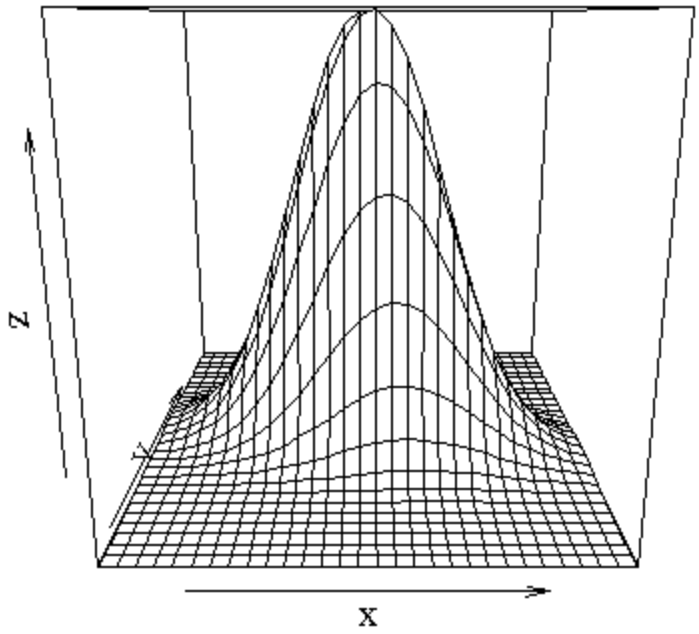
产生图 6。对不同性别分别绘制了体重对身高的散点图。如果 z 是一个数值型变量，则 coplot() 先对 z 的取值分组，然后对 z 的每一组取值分别绘图。甚至可以用如 coplot(y~x | x1+x2)表示对 x1 和 x2 的每一水平组合绘图。coplot()和 pairs()函数缺省绘制散点图，但可以用一个 panel= 参数指定其它的低级绘图函数，如 lines，panel.smooth 等。

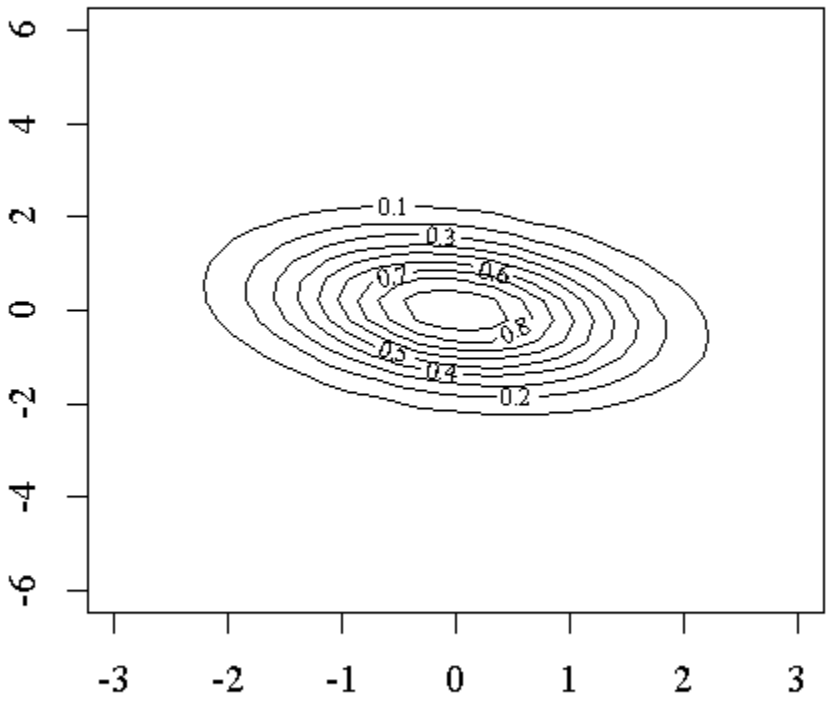
tsplot(x)绘制时间序列曲线图。多个参数时 tsplot(x1, x2, ...)表示绘制多条曲线，自动统一曲线取值范围。如果参数非时间序列对象则以下标 1，2，3 等为横坐标绘图。

qqnorm(x), qqline(x), qqplot(x,y)作分位数－分位数图。qqnorm(x)对向量 x 作正态概率（纵轴为次序统计量值，横轴为对应该次序统计量的标准正态分布分位数值）。qqline(x) 除作 qqnorm(x)图之外还画一条拟合曲线。qqplot(x,y)把 x 和 y 的次序统计量分别画在 x 轴和 y 轴以比较两个变量的分布。

hist(x)作向量 x 的直方图。缺省时自动确定分组，也可以用 nclass=参数指定分组个数，或者用 breaks=参数指定一个分组点向量。如果指定了 prob=T 则纵轴显示密度估计。

S 也可以作三维图或等值线图，函数为 persp()和 contour()，见图 7 和图 8。





高级图形函数的常用选项

高级图形函数有一些共同的选项，作为函数的可选参数（自变量）。例如：

> plot(x)

> plot(x, main="Graph of x")

其中的 main 就是一个可选参数，用来指定图形的标题。没有此选项时图形就没有标题。这样的选项还有：

add=T	使函数向低级图形函数那样不是开始一个新图形而是在原图基础上添加。
axes=F	暂不画坐标轴，随后可以用 axis()函数更精确地规定坐标轴的

	画法。缺省值是 axes=T，即有坐标轴。
log="x" log="y" log="xy"	把 x 轴，y 轴或两个坐标轴用对数刻度绘制。
type= • type="p" • type="l" • type="b" • type="o" • type="h" • type="s" • type="S" • type="n"	规定绘图方式： • 绘点 • 画线 • 绘点并在中间用线连接 • 绘点并画线穿过各点 • 从点到横轴画垂线 • 阶梯函数；左连续 • 阶梯函数；右连续 • 不画任何点、线，但仍画坐标轴并建立坐标系，适用于后面用低级图形函数作图。
xlab="字符串" ylab="字符串" main="字符串" sub="字符串"	定义 x 轴和 y 轴的标签。缺省时使用对象名。 图形的标题。 图形的小标题，用较小字体画在 x 轴下方。

低级图形函数

高级图形函数可以迅速简便地绘制常见类型的图形，但是，某些情况下你可能希望绘制一些有特殊要求的图形。比如，你希望坐标轴按照自己的设计绘制，在 已有的图上增加另一组数据，在图中加入一行文本注释，绘出多个曲线代表的数据的标签，等等。低级图形函数让你在已有的图的基础上进行添加。

常用的低级图形函数罗列如下：

points(x,y) lines(x,y)	在当前图形上叠加一组点或线。可以使用 plot()的 type=参数来指定绘制方法，缺省时 points() 画点，lines()画线。
text(x,y,	在由坐标 x 和 y 给出的位置标出由 labels 指定的字符串。labels

labels, ...)	可以是数值型或字符型的向量，labels[i]在 x[i]，y[i]处标出。
abline(a, b) abline(h=y) abline(v=x) abline(*lm.obj*)	在当前图形上画一条直线。两个参数 a, b 分布给出截距和斜率。指定 h=参数时绘制水平线，指定 v=参数时绘制垂直线。以一个最小二乘拟合结果 lm.obj 作为参数时由 lm.obj 的 $coefficients 成员给出直线的截距和斜率。
polygon(x, y, ...)	以由向量 x 给出的横坐标和向量 y 给出的纵坐标为顶点绘制多边形。可以用 col=参数指定一个颜色填充多边形内部。
legend(x, y, legend, ...) legend(, angle=v) legend(, density=v) legend(, fill=v) legend(, col=v) legend(, lty=v) legend(, pch=v) legend(, marks=v)	legend 函数用来在当前图形的指定坐标位置绘制图例。图例的说明文字由向量 legend 提供。至少下面的 v 值要给出以确定要对什么图例进行说明，v 是长度与 legend 相同的向量。 angle 参数指定几种阴影斜角。 density 参数指定几种阴影密度。 fill 参数指定几种填充颜色。 col 参数指定几种颜色。 lty 参数指定几种线型。 pch 参数指定几种散点符号。为字符型向量。 marks 参数也指定几种散点符号，但使用散点符号数值代号，为数值型向量。
title(main, sub)	绘制由 main 指定的标题和由 sub 指定的小标题。
axis(side, ...)	绘制一条坐标轴。这之前的绘图函数必须已经用 axes=F 选项抑制了自动的坐标轴。参数 side 指定在哪一边绘制坐标轴，取值为 1 到 4，1 为下边，然后逆时针数。可以用 at=参数指定刻度位置，用 labels 参数指定刻度处的标签。

低级图形函数一般需要指定位置信息，其中的坐标指的是所谓用户坐标，即前面的高级图形函数所建立的坐标系中的坐标。坐标可以用两个向量 x 和 y 给出，也可以由一个两列的矩阵给出。如果交互作图可以用下面介绍的 locator()函数来交互地从图形中直接输入坐标位置。

交互图形函数

S 的低级图形函数可以在已有图形的基础上添加新内容，另外，S 还提供了两个函数 locator 和 identify 可以让用户通过在图中用鼠标点击来确定位置。

函数 locator(n, type)运行时会停下来等待用户在图中点击，然后返回图形中鼠标点击的位置的坐标。等待点击时用鼠标中键点击可以选择停止等待，立即返回。参数 n 指定点击多少次后自动停止，缺省为 500 次；参数 type 如果使用则可指定绘点类型，与 plot()函数中的 type 参数用法相同，在鼠标点击处绘点（线、垂线，等等）。locator()的返回值是一个列表，有两个变量（元素）x 和 y，分别保存点击位置的横坐标和纵坐标。

例如，为了在已经绘制的曲线图中找一个空地方标上一行文本，只要使用如下程序：

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

text()函数的 adj 参数用一个数字表示文本串相对于给定的坐标的画法，adj=0 表示给定坐标为文本串左侧的坐标，adj=1 表示给定坐标为文本串右侧的坐标，adj=0.5 表示给定坐标为文本串中间的坐标。

函数 identify(x, y, labels)在运行时也会停下来等待用户点击，直到按了鼠标中键，任何返回用户在图形中用鼠标点击的点的序号，点击时对点击的点加标签。参数 x 和 y 给出 要识别的各个点的坐标。labels 参数指定点击某个点时要在旁边绘制的文本标签，缺省时标出此点的序号，如果只需要返回值而不想画任何标记则可以在调用 此函数时加一个 plot=F 参数。注意 identify()与 locator()不同，locator()返回图中任意点击位置的坐标，而 identify() 只返回离点击位置最近的点的序号。

例如，我们在向量 x 和 y 中有若干个点的坐标，运行如下程序：

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

这时显示转移到图形窗口，进入等待状态，用户可以点击图中特别的点，该点的序号就会在旁边标出。为了结束，只要单击鼠标中键或单击右键并选择停止。返回结果为你点击的各个点的序号：

[1] 4 6 7 8

图形参数的使用

前面我们已经看到了如何用 main=，xlab=等参数来规定高级图形函数的一些设置。在实际绘图，特别是绘制用于演示或出版的图形时，S 用缺省设 置绘制的图形往往不能满足我们的要求。但是，S 提供了一系列所谓图形参数，通过使用图形参数

可以修改图形显示的所有各方面的设置。图形参数包括关于线型、 颜色、图形排列、文本对齐方式等各种设置。每个图形参数有一个名字，比如 col 代表颜色，取一个值，比如 col="red"是红色。每个图形设备有一套单 独的图形参数。

设置图形参数分为两种：永久设置与临时设置。永久设置使用 par()函数进行设置，设置后在退出前一直保持有效；临时设置则是在图形函数中加入图形参数，如上面的例子：

```
> text(locator(1), "Normal density", adj=0)
```
中的 adj 参数。

par()函数用来访问或修改当前图形设备的图形参数。如果不带参数调用，如：

```
> par()
$adj
[1] 0.5

$ann
[1] 1

……

$tcl
[1] -0.5
```
结果为一个列表，列表的各元素名为图形参数的名字，元素值为相应图形参数的取值。

如果调用时指定一个图形参数名的向量作为参数，则只返回被指定的图形参数的列表：

```
> par(c("col", "lty"))
$col
[1] "black"

$lty
[1] "solid"
```

调用时指定名字为图形参数名的有名参数，则修改指定的图形参数，并返回原值的列表：

```
> oldpar <- par(col=4, lty=2)
> oldpar
$col
[1] "black"

$lty
[1] "solid"
```

因为用 par()修改图形参数是保持到退出以前都有效的，而且即使是在函数内此修改仍是全局的，所以我们可以利用如下的惯用法，在完成任务后恢复原来的图形参数：

```
> oldpar <- par(col=4, lty=2)
……（需要修改图形参数的绘图任务）
> par(oldpar)     # 恢复原始的图形参数
```

除了象上面那样用 par()函数永久修改图形参数，我们还可以在几乎任何图形函数中指定图形参数作为有名参数，这样的修改是临时的，只对此函数起作用。例如：

```
> plot(x, y, pch="+")
```

就用图形参数 pch 指定了绘散点的符号为加号。这个设定只对这一张图有效，对以后的图形没有影响。

图形参数详解

鉴于绘制有特殊需要的图形是 S 的一个强项，而使用图形参数是完成此类任务的重要手段，我们在这里较详细地介绍 S 的各种图形参数。这些图形参数可以大体上分为以下的几个大类，我们将分别介绍：

- 图形元素控制
- 坐标轴与坐标刻度
- 图形边空
- 一页多图

一、图形元素

图形由点、线、文本、多边形等元素构成。下列的图形参数用来控制图形元素的绘制细节：

`pch="+"`	指定用于绘制散点的符号。绘制的点往往略高于或低于指定的坐标位置，只有 pch="."没有这个问题。
pch=4	如果 pch 的值为从 0 到 18 之间的一个数字，将使用特殊的绘点符号。下例可以显示所有特殊绘点符号：

	`> plot(c(0, 100), c(0, 100), type="n", axes=F, xlab='', ylab='')`
	`> legend(10,90, as.character(0:9), pch=0:9)`
	`> legend(50,90, as.character(10:18), pch=10:18)`

lty=2	指定画线用的线型。缺省值 lty=1 是实线。从 2 开始是各种虚线。

lwd=2	指定线粗细，以标准线粗细为单位。这个参数影响数据曲线的线宽以及坐标轴的线宽。下例绘制正弦曲线图：
	
	`> oldpar <- par(lwd=2)`
	`> x <- (0:100)/100*2*pi`
	`> plot(x, sin(x), type="l", axes=F)`
	`> abline(h=0)`
	`> abline(v=0)`
	`> par(oldpar)`

col=2	指定颜色，可应用于绘点、线、文本、填充区域、图象。颜色值也可以用象"red"，"blue" 这样的颜色名指定。
font=2	用来指定字体的整数。一般 font=1 是正体，2 是 **黑体**，3 是 *斜体*，4 是 ***黑斜体***。

`font.axis`
`font.lab`
`font.main`
 	
font.sub
	分别用来指定坐标刻度、坐标轴标签、标题、小标题所用的字体。

adj=-0.1	指定文本相对于给定坐标的对齐方式。取 0 表示左对齐，取 1 表示右对齐，取 0.5 表示居中。此参数的值实际代表的是出现在给定坐标左边的文本的比例，所以 adj=-0.1 的效果是文本出现在给定坐标位置的右边并空出相当于文本 10%长度的距离。
cex=1.5	指定字符放大倍数。

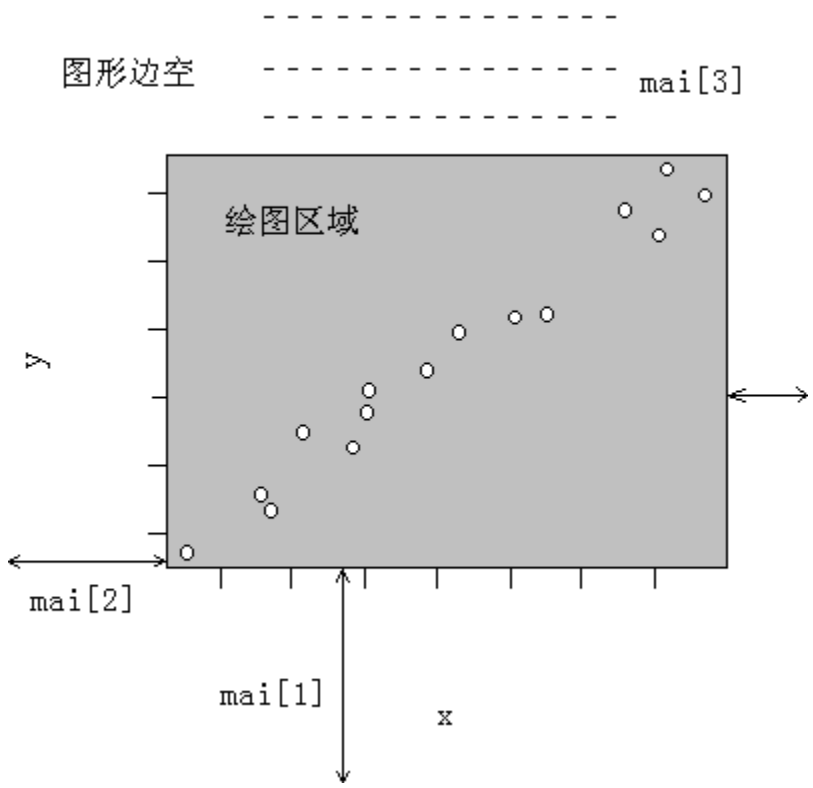
二、坐标轴与坐标刻度

许多高级图形带有坐标轴，还可以先不画坐标轴然后用 axis()单独加。函数 box()用来画坐标区域四周的框线。

坐标轴包括三个部件：轴线（用 lty 可以控制线型），刻度线，刻度标签。它们可以用如下的图形参数来控制：

lab=c(5, 7, 12)	第一个数为 x 轴希望画几个刻度线，第二个数为 y 轴希望画几个刻度线，这两个数是建议性的；第三个数是坐标刻度标签的宽度为多少个字符，包括小数点，这个数太小会使刻度标签四舍五入成一样的值。
las=1	坐标刻度标签的方向。0 表示总是平行于坐标轴，1 表示总是水平，2 表示总是垂直于坐标轴。
mgp=c(3,1,0)	坐标轴各部件的位置。第一个元素为坐标轴位置到坐标轴标签的距离，以文本行高为单位。第二个元素为坐标轴位置到坐标刻度标签的距离。第三个元素为坐标轴位置到实际画的坐标轴的距离，通常是 0。
tck=0.01	坐标轴刻度线长度，单位是绘图区域大小，值为占绘图区域的比例。tck 小于 0.5 时 x 轴和 y 轴的刻度线将统一到相同的长度。取 1 时即画格子线。取负值时刻度线画在绘图区域的外面。
xaxs="s" yaxs="d"	控制 x 轴和 y 轴的画轴方法。 取值为"s"（即 standard）或"e"（即 extended）的时候数据范围控制在最小刻度和最大刻度之间。取"e"时如果有数据点十分靠近边缘轴的范围会略微扩大。这种画轴方式有时会在轴的一边留下太大的空白。 取值为"i"（即 internal）或"r"（此为缺省）使得刻度线都落在数据范围内部，而"r"方式所留的边空较小。 取值设为"d"时会锁定此坐标轴，后续的图形都使用与它完全相同的坐标轴，这在要生成一系列可比较的图形的时候是有用的。要解除锁定需要把这个图形参数设为其它值。

三、图形边空



S 中一个单独的图由绘图区域（绘图的点、线等画在这个区域中）和包围绘图区域的边空组成，边空中可以包含坐标轴标签、坐标轴刻度标签、标题、小标题等，绘图区域一般被坐标轴包围。见图 9。

边空的大小由 mai 参数或 mar 参数控制，它们都是四个元素的向量，分别规定下方、左方、上方、右方的边空大小，其中 mai 取值的单位是英寸，而 mai 的取值单位是文本行高度。例如：

> par(mai=c(1, 0.5, 0.5, 0))

> par(mar=c(4, 2, 2, 1))

这两个图形参数不是独立的，设定一个会影响另一个。S 缺省的图形边空常常太大，以至于有时图形窗口较小时边空占了整个图形的很大一部分。通常我们可以取消右边空，并且在不用标题时可以大大缩小上边空。例如下例可以生成十分紧凑的图形：

> oldpar <- par(mar=c(2,2,1,0.2))

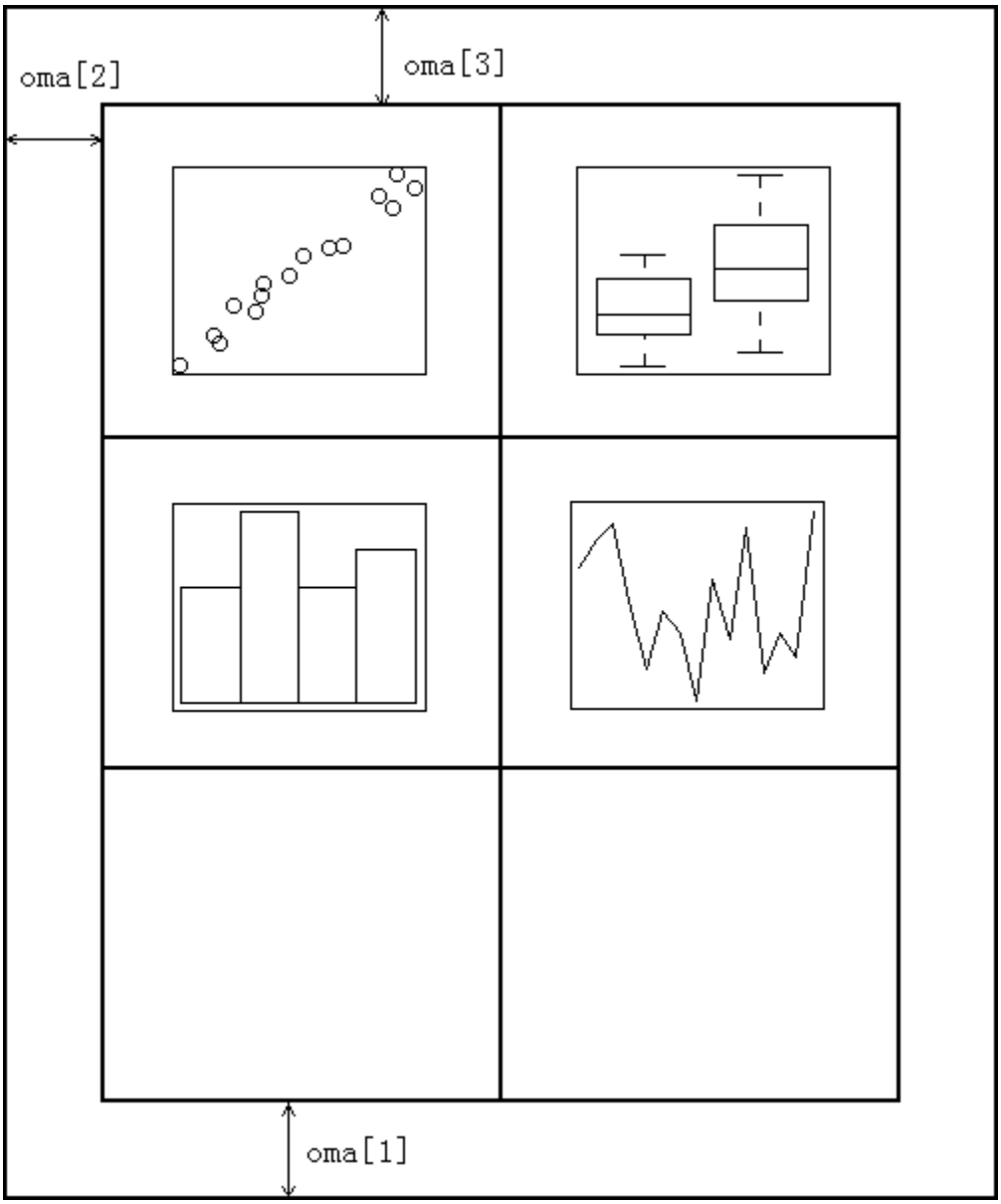
> plot(x,y)

在一个页面上画多个图时边空自动减半，但我们往往还需要进一步减小边空才能使多个图有意义。

四、一页多图

R 可以在同一页面开若干个按行、列排列的窗格，在每个窗格中可以作一幅图。每个图有自己的边空，而所有图的外面可以包一个“外边空”，见图 10。

一页多图用 mfrow 参数或 mfcol 参数规定，如：



> par(mfrow=c(3,2))

表示同一页有三行两列共六个图，而且次序为按行填放。类似地，

> par(mfcol=c(3,2))

规定相同的窗格结构，但是次序为按列填放，即先填满第一列的三个再填第二列。要取消一页多图只要再运行

> par(mfrow=c(1,1))

即可。

缺省时无外边空。为了规定外边空大小，可以用 omi 参数或 oma 参数。omi 参数使用英寸为单位，oma 参数以文本行高为单位，两个参数均为四个元素的向量，分别给出下、左、上、右方的边空大小。如：

> par(oma=c(2,0,3,0))

函数 mtext 用来在外边空加文字标注。其用法为

mtext(text, side = 3, line = 0, outer = FALSE)

其中 text 为要加的文本内容，side 表示在哪一边写（1 为下，2 为左，3 为上，4 为右），line 表示边空从里向外数的第几行，最里面的一行是第 0 号，outer=TRUE 时使用外边空，否则会使用当前图的边空。例如：

```
> par(mfrow=c(2,2), oma=c(0,0,3,0), mar=c(2,1,1,0.1))
> plot(x);plot(y);boxplot(list(x=x,y=y));plot(x,y)
> mtext("Simulation Data", outer=T, cex=1.5)
```

在多图环境中还可以用 mfg 参数来直接跳到某一个窗格，比如

> par(mfg=c(2,2,3,2))

表示在三行两列的多图环境中直接跳到第二行第二列位置。mfg 参数的后两个表示多图环境的行、列数，前两个表示要跳到的位置。

可以不使用多图环境而直接在页面中的任意位置产生一个窗格来绘图，参数为 fig，如：

> par(fig=c(4,9,1,4)/10)

此参数为一个向量，分别给出窗格的左、右、下、上边缘的位置，取值为占全页面的比例，比如上面的例子在页面的右下方开一个窗格作图。

图形设备

S 作图支持各种图形设备，其中常用的是显示器和 PostScript 打印机。在一个 S 运行期间可以有多个图形设备同时存在。在 R 中，用

> x11()

打开图形窗口绘图，在 S-PLUS 中，用

> win.graph()

打开图形窗口绘图。再次调用这样的函数将打开第二个图形窗口。用

> dev.list()

可显示以打开的图形设备的列表。

要关闭一个图形设备，用

> dev.off()

这可以使得图形得以完成，例如对于 postscript 设备关闭设备时可完成打印或存盘。用 graphics.off() 函数可以关闭所有打开的图形设备。

MS Windows 下的 R 可以把显示窗口中的图形复制到剪贴板或存为各种格式的图形文件，包括 WMF、PostScript、PNG、BMP、JPEG，这样我们可以用 R 生成所需图形然后存为需要的格式。MS Windows 下的 S-PLUS 也具有类似功能。

各版本的 R 和 S-PLUS 都支持生成 PostScript 图形的功能，生成的图形可以直接用于 LaTeX 排版。如果用 MS Word 排版则可把屏幕图形存为 WMF 等格式。生成 PostScript 文件的设备可以用如下函数打开：

> postscript(file="result1.ps", horizontal=FALSE, width=5, height=3)

这时用图形命令生成一个页面的图形，然后用 dev.off()关闭设备，则可生成文件 result1.ps 。postscript()函数中 horizotal 参数指定是否将图旋转 90 度使得 x 轴平行于纸的长边，width 和 height 规定图的宽和高，单位是英寸。

在打开了多个图形设备后可以用 dev.set()函数来选择当前设备，dev.next()和 dev.prev() 分别返回下一个和上一个图形设备。比如 dev.set(dev.prev())选择上一个图形设备。

S 的对象

S 是一种面向对象的语言。一般说来，S 的对象包含了若干个元素作为其数据，这些元素的个数叫做此对象的长度（length），这些元素的共同的类型 叫做此对象的模式（mode）。另外，对象还可以包含一些特殊数据，称为属性（attribute），如列表的每一个成员（元素）都可以有变量名，这些变 量名组成的字符型向量为此列表的 names 属性。

S 的面向对象能力依赖于对类属性的使用。S 的对象有的是简单对象，这样的对象没有类（class ）属性，可以认为其类为缺省类（default）；有的是属于某一类的对象，这些对象有一个类（class）属性，同类的对象具有相同的特征，可以为同类的对象定义针对这一类的特殊操作（如显示、绘图），在面向对象术语中叫做方法。比如，向量是简单对象，它没有类属性；数据框也是对象，但是数据框有一个类属性 class=data.frame。

固有属性：mode 和 length

S 对象都有两个基本的属性（attribute）：类型（mode）属性和长度（length）属性。

S 对象可分为单纯的（atomic）和复合的（recursive）两种，单纯对象的所有元素都是同一种基本类型（如数值、字符串），元素不再是对 象，这样的对象的类型（mode）有 logical （逻辑型）、numeric（数值型）、complex（复数型）、character（字符型）等等；复合对象的元素可以是不同类型，每一个元素是一个 对象，这样的对象最常用的是列表。例如，向量（vector）是单纯对象，它的所有元素都必须是相同类型，数值型向量的所有元素必须为数值型，字符型向量 的所有元素必须为字符型；列表（list）是复合对象，类型（mode）为列表（list），列表的每一个元素（变量）都可以是一个 S 对象，比如列表元素 可以为一个数，一个字符串，一个向量，甚至一个列表。

S 对象有一种特别的 null（空值型）型，只有一个特殊的 NULL 值为这种类型，表示没有值（不同于 NA，NA 是一种特殊值，而 NULL 根本没有对象值）。

为了判断对象的类型，S 定义了许多个类似于 is.numeric()这样的函数。比如，is.numeric(x) 用来检验对象 x 是否数值型，返回一个逻辑型标量结果；is.character() 检验对象是否字符型，等等。

长度属性表示 S 对象元素的个数，比如 length(2:4)等于 3。注意向量允许长度为 0，数值型向量长度为零表示为 numeric()或 numeric(0)，字符型向量长度为零表示为 character()或 character(0) 。

S 可以强制进行类型转换，例如

```
> z <- 0:9
> digits <- as.character(z)
> d <- as.numeric(digits)
```

第二个赋值把数值型的 z 转换为字符型的 digits。第三个赋值把 digits 又转换为了数值型的 d，这时 d 和 z 是一样的了。S 还有许多这样的以 as.开头的类型转换函数。

S 允许对超出对象长度的下标赋值，这时对象长度自动伸长以包括此下标，未赋值的元素取缺失值（NA），例如：

```
> x <- numeric()
> x[3] <- 100
> x
[1]  NA  NA 100
```

要缩短对象的长度又怎么办呢？只要给它赋一个子集就可以了。例如：

```
> x <- 1:4
> x <- x[1:2]
> x
[1] 1 2
```

访问对象属性

对象属性是对象包含的数据中除元素以外的特殊数据，每个属性有一个属性名，有一个属性值。S 定义了两个函数 attributes 和 attr 来访问对 象的属性。attributes(object)返回对象 object 的各特殊属性组成的列表，其中不包括固有属性 mode 和 length。例如：

```
> x <- c(apple=2.5, orange=2.1)
> attributes(x)
$names
[1] "apple"  "orange"
```

可以用 attr(object, name)的形式存取对象 object 的名为 name 的属性。例如：

```
> attr(x, "names")
[1] "apple"  "orange"
```

也可以把 attr()函数写在赋值的左边以改变属性值或定义新的属性，例如：

```
> attr(x, "names") <- c("apple", "grapes")
> x
 apple grapes
   2.5    2.1
```

```
> attr(x, "type") <- "fruit"
> x
 apple grapes
   2.5    2.1
attr(,"type")
[1] "fruit"
> attributes(x)
$names
[1] "apple"  "grapes"

$type
[1] "fruit"
```

这种对一个函数赋值的语法是在其它语言中极为少见的，而 S 中则经常使用这样的写法。实际上，attr(x, "names")在这里不应该看成是一个函数值，而应该看成是用来保存对象 x 的 names 属性的变量名。

对象的类

S 用类（class）属性来支持面向对象的编程风格。对象的类属性区分对象的类，对于同一类的对象可以定义一组特殊操作，这一点和其它面向对象语言 类似。面向对象风格的最重要的特点就是数据抽象与封装。所谓数据抽象与封装是指对象的用户要访问或修改对象只能通过对象提供的服务来进行，用户不能看到对 象内部的实现细节。这样用户不会直接修改对象的数据从而保护了数据的完整性，而且用户只需要知道对象提供了哪些服务，即使对象内部的实现改变了，只要接口 不变则用户程序不必改变。这样的做法可以提高程序的安全性和可重用性。

常见的面向对象语言一般先定义一个类，这个类定义了一些数据结构，然后有一些函数叫做“方法”可以操作这些数据。所谓对象，是由某个类生成的实例， 其数据结构由所属的类定义，而实际存储的数据则是属于对象本身的。对象拥有其所属类的所有方法，方法在调用时操作的是属于这个对象的数据。

S 也支持面向对象编程，但是做法与常见的面向对象语言有很大差别。S 对象的类由其类（class ）属性指定，每一个类都可以定义本类的服务，服务以函数形式定义，调用格式为“函数名（对象，其它自变量）”。可见 S 的类机制是比较松散的，它不象常见的 面向对象语言那样必须先定义类的所有数据结构与方法，而是可以随时定义函数作为类对象的服务。另外，S 还定义了一系列的所谓“通用函数（general functions）”，通用函数也是对象提供的服务，但不同类的对象都可以使用相同的通用函数名字调用，同一个通用函数可以针对不同类的对象起到相似的作用。用户只需要记忆很少的几个通用函数的名字，就可以对几乎所有对象调用这些函数。比如，通用 print()函数用来显示对象，它可以显示向量和矩阵， 但显示方法不同；通用函数 plot()函数用来画对象的图形，对一个向量画图 plot()画散点图，纵轴为各元素值，横轴为元素下标；对一个时间序列对象 画图 plot()将画一条时间序列曲线，并用年月等标记时间轴。

S 的每一个通用函数实际是一组函数，有一个共同的名字，在调用时根据自变量的类（class ）的不同决定调用一组中的哪一个函数。例如，对向量 x 调用 print(x)实际调用的是 print.default(x) ，对数据框 x 调用 print(x)则实际调用的是 print.data.frame(x)。如果自变量没有类属性，或者此通用函数没有为此类自变量设计特 殊的操作，通用函数总有一个缺省方法可以调用（如 print.default）。通用函数针对某一类的对象的特殊函数的命名为“通用函数名.类名() ”。

对某一种类的对象有特殊操作的通用函数可以有很多个，比如，对 data.frame 类的对象定义了特殊操作的通用函数就有：

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

等等。如果对 data.frame 类的对象 d 调用 plot(d)，实际调用的函数是 plot.data.frame(d) 。要列出所有对某类有特殊操作的通用函数，可以用

```
> methods(class="data.frame")
 [1] "Math.data.frame"           "Ops.data.frame"
……
 [27] "summary.data.frame"       "t.data.frame"
 [29] "transform.data.frame"     "xpdrows.data.frame"
```

其中 t.data.frame 就是调用 t(d)时实际调用的函数。

也可以列出某通用函数的对各类的特殊定义，例如：

```
> methods(plot)
 [1] "plot.data.frame" "plot.default"    "plot.density"
 [4] "plot.factor"     "plot.formula"    "plot.function"
 [7] "plot.lm"         "plot.mlm"        "plot.mts"
[10] "plot.new"        "plot.ts"         "plot.window"
[13] "plot.xy"
```

比如，plot.factor 是对因子对象调用 plot()函数是实际调用的函数。

为了暂时去掉一个有类的对象的 class 属性，可以使用 unclass(object)函数。

S 统计模型简介

这一节我们简单介绍 S 的统计模型。S 中实现了几乎所有常见的统计模型，而且多种模型可以用一种统一的观点表示和处理。这方面 S-PLUS 较全面，它 实现了许

多最新的统计研究成果，R 因为是自愿无偿工作所以统计模型部分还相对较欠缺。事实上，许多统计学家的研究出的统计算法都以 S-PLUS 程序发表， 因为 S 语言是一种特别有利于统计计算编程的语言。

学习这一节需要我们具备线型模型、线型回归、方差分析的基本知识。

统计模型的表示

很多统计模型可以用一个线型模型来表示：

$$y_i = \sum_{j=0}^{p} \beta_j x_{ij} + e_i, \quad e_i \sim \text{iid}\, N(0, \sigma^2), \quad i = 1, 2, \ldots, n$$

写成矩阵形式为

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{e}$$

其中 $\boldsymbol{y}$ 为因变量， $\boldsymbol{X}$ 为模型矩阵或称设计阵，各列为 $\boldsymbol{x}_0$， $\boldsymbol{x}_1$，...， $\boldsymbol{x}_p$ 等各自变量，其中 $\boldsymbol{x}_0$ 常常是一列 1，定义一个模型截距项。

在 S 中模型是一种对象，其表达形式叫做一个公式，我们先举几个例子来看一看。假定 y，x，x0，x1，x2，…是数值型变量，X 是矩阵，A，B，C，…是因子。

y ~ x y ~ 1 + x	两个式子都表示 y 对 x 的简单一元线型回归。第一个式子带有隐含的截距项，而第二个式子把截距项显式地写了出来。
y ~ -1 + x y ~ x - 1	都表示 y 对 x 的通过原点的回归，即不带截距项的回归。
log(y) ~ x1 + x2	表示 log(y)对 x1 和 x2 的二元回归，带有隐含的截距项。
y ~ poly(x, 2) y ~ 1 + x + I(x^2)	表示 y 对 x 的一元二次多项式回归。第一种形式使用正交多项式，第二种形式直接使用 x 的各幂次。
y ~ X + poly(x,2)	因变量为 y 的多元回归，模型矩阵包括矩阵 X，以及 x 的二次多项式的各项。

y ~ A	一种方式分组的方差分析，指标为 y，分组因素为 A。
y ~ A + x	一种方式分组的协方差分析，指标为 y，分组因素为 A，带有协变量 x。

`y ~ A*B`
`y ~ A + B + A:B`
`y ~ B %in% A`
 	
y ~ A/B
	非可加两因素方差分析模型，指标为 y，A，B 是两个因素。前两个公式表示相同的交叉分类设计，后两个公式表示相同的嵌套分类设计。

y ~ (A + B + C)^2 y ~ A*B*C - A:B:C	表示三因素试验，只考虑两两交互作用而不考虑三个因素间的交互作用。两个公式是等价的。

`y ~ A * x`
`y ~ A / x`
 	
y ~ A / (1 + x) - 1
	都表示对因子 A 的每一水平拟合 y 对 x 的线型回归，但三个公式的编码方式不同。最后一种形式对 A 的每一水平都分别估计截距项和斜率项。

y ~ A*B + Error(C)	表示有两个处理因素 A 和 B，误差分层由因素 C 确定的设计

在 S 中~运算符用来定义模型公式。一般的线型模型的公式形式为

因变量 ~ [±]第一项 [±]第二项 [±]第三项 [±]…

其中因变量可以是向量或矩阵，或者结果为向量或矩阵的表达式。 [±]是加号+或者减号－，表示在模型中加入一项或去掉一项，第一项前面如果是加号可以省略。

公式中的各项可以取为：

- 一个值为向量或矩阵的表达式，或 1。
- 一个因子
- 一个“公式表达式”，由“公式运算符”把因子、向量、矩阵连接而成。

每一项定义了要加入模型矩阵或从模型矩阵中删除的若干列。一个 1 表示一个截距项列，除非显式地删除总是隐含地包括在模型公式中。

“公式运算符”的定义和 Glim、Genstat 软件中的定义类似，不过那里的“.”运算符这里改成了“:”，因为在 S 中句点是名字的合法字符。下表列出了各运算符的简要说明。

运算符	含义
Y ~ *M*	*Y* 作为因变量由 *M* 解释。
$M_1 + M_2$	加入 M_1 和 M_2
$M_1 - M_2$	加入 M_1 但去掉 M_2 指定的项
$M_1 : M_2$	M_1 和 M_2 的张量积。如果两项都是因子，则为因子的交互作用。
M_1 %in% M_2	与 $M_1 : M_2$ 类似但模型矩阵编码方式不同。
$M_1 * M_2$	等于 $M_1 + M_2 + M_1 : M_2$
M_1 / M_2	等于 $M_1 +$ M_2 %in% M_1
M^ *n*	*M* 中所有各项以及所有到 *n* 阶为止的交互作用项。
I(*M*)	将 *M* 隔离，使得 *M* 中的运算符按照原来的算术运算符解释而不是按公式运算符解释。表达式 *M* 的结果作为公式的一项。

注意在函数调用的括号内的表达式按普通四则运算解释。函数 I()可以把一个计算表达式封装起来作为模型的一项使用。

注意 S 的模型表示只给出了因变量和自变量及自变量间的关系，这样只确定了线型模型的模型矩阵，而模型参数向量是隐含的，并没有的模型公式中体现出来。这种做法适用于线性模型，但不具有普遍性，例如非线性模型就不能这样表示。

线性回归模型

拟合普通的线性模型的函数为 lm()，其简单的用法为：

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

其中 data.frame 为各变量所在的数据框，formula 为模型公式，fitted.model 是线性模型拟合结果对象（其 class 属性为 lm）。例如：

> mod1 <- lm(y ~ x1 + x2, data=production)

可以拟合一个 y 对 x1 和 x2 的二元回归（带有隐含的截距项），数据来自数据框 production 。拟合的结果存入了对象 mod1 中。注意不论数据框 production 是否以用 attach()连接入当前运行环境都可被 lm()使用。lm()的基本显示十分简练：

```
> mod1

Call:
lm(formula = y ~ x1 + x2, data = production)

Coefficients:
(Intercept)           x1           x2
  0.0122033    2.0094758   -0.0005314
```

只显示了调用的公式和参数估计结果。

提取信息的通用函数

lm()函数的返回值叫做模型拟合结果对象，本质上是一个具有类属性值 lm 的列表，有 model 、coefficients、residuals 等成员。lm()的结果显示十分简单，为了获得更多的拟合信息，可以使用对 lm 类对象有特殊操作的通用函数，这些函数包括：

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

下表给出了 lm 类（拟合模型类）常用的通用函数的简单说明。

通用函数	返回值或效果
anova(对象 1，对象 2)	把一个子模型与原模型比较，生成方差分析表。
coefficients(对象)	返回回归系数（矩阵）。可简写为 coef(对象)。
deviance(对象)	返回残差平方和，如有权重则加权。
formula(对象)	返回模型公式。

plot(对象)	生成两张图，一张是因变量对拟合值的图形，一张是残差绝对值对拟合值的图形。
predict(对象， newdata=数据框) predict.gam(对象， newdata=数据框)	有了模型拟合结果后对新数据进行预报。指定的新数据必须与建模时用的数据具有相同的变量结构。函数结构为对数据框中每一观测的因变量预报结果（为向量或矩阵）。 predict.gam()与 predict()作用相同但适用性更广，可应用于 lm、glm 和 gam 的拟合结果。比如，当多项式基函数用了正交多项式时，加入了新数据导致正交多项式基函数改变，用 predict.gam() 函数可以避免由此引起的偏差。
print(对象)	简单显示模型拟合结果。一般不用 print()而直接键入对象名来显示。
residuals(对象)	返回模型残差（矩阵），若有权重则适当加权。可简写为 resid(对象)。
summary(对象)	可显示较详细的模型拟合结果。

方差分析

方差方差分析是研究取离散值的因素对一个数值型指标的影响的经典工具。S 进行方差分析的函数是 aov()，格式为 aov(公式，data=数据框)，用法与 lm()类似，提取信息的各通用函数仍有效。

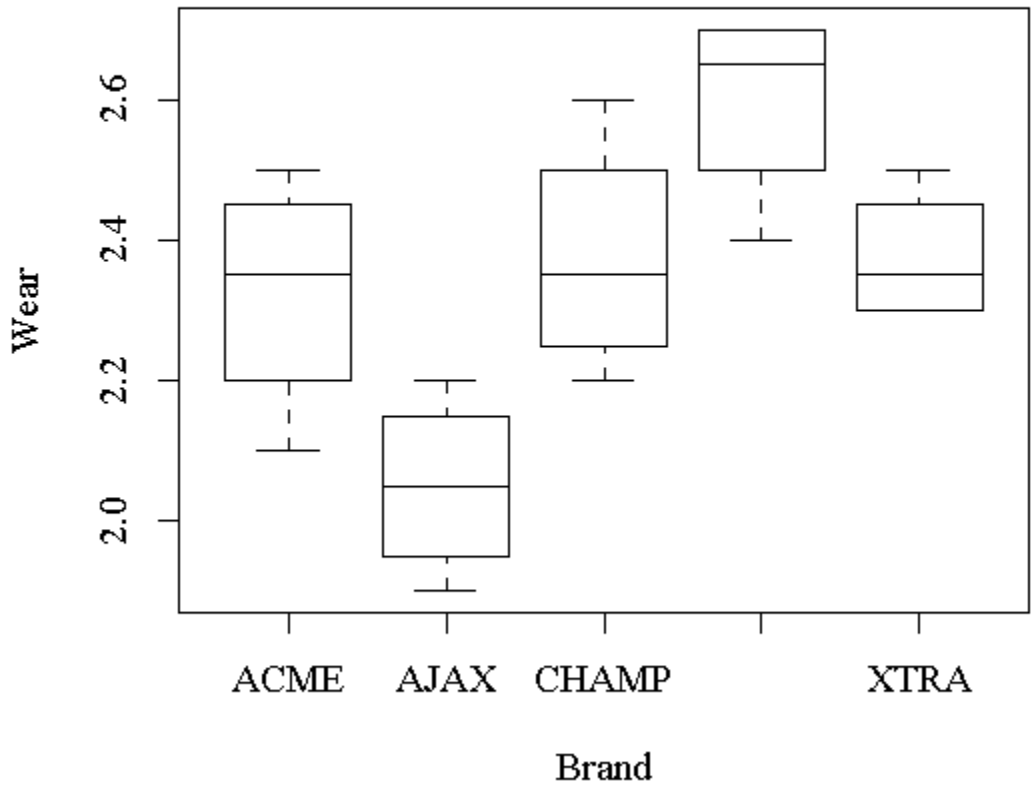
我们以前面用过的不同牌子木板磨损比较的数据为例。假设 veneer 数据框保存了该数据：

```
> veneer
   Brand Wear
1   ACME  2.3
2   ACME  2.1
3   ACME  2.4
4   ACME  2.5
5  CHAMP  2.2
6  CHAMP  2.3
7  CHAMP  2.4
8  CHAMP  2.6
9   AJAX  2.2
10  AJAX  2.0
11  AJAX  1.9
12  AJAX  2.1
13 TUFFY  2.4
14 TUFFY  2.7
```

```
15 TUFFY  2.6
16 TUFFY  2.7
17  XTRA  2.3
18  XTRA  2.5
19  XTRA  2.3
20  XTRA  2.4
```

首先我们把每个牌子的木板的磨损情况画盒形图并且放在同一页面中，作图如下：

```
plot(Wear ~ Brand, data=veneer)
```



见图 11。这种图可以直观地比较一个变量在多个组的分布，或者比较几个类似的变量。从图中可以看出，AJAX 牌子较好，TUFFY 较差，其它三个牌子差别不明显。

为了检验牌子这个因素对指标磨损量有无显著影响，只要用 aov()函数：

```
> aov.veneer <- aov(Wear ~ Brand, data=veneer)
> summary(aov.veneer)
            Df  Sum Sq Mean Sq F value   Pr(>F)
Brand        4 0.61700 0.15425   7.404 0.001683 **
Residuals   15 0.31250 0.02083
```

```
---
Signif. codes:  0  `***'  0.001  `**'  0.01  `*'  0.05  `.'  0.1  ` '
1
```
可见因素是显著的。

统计分析实例

下面我们以 7.1.2 中的那个学生班的情况为例进行一些分析。我们希望了解体重、身高、年龄、性别等变量的基本情况及互相之间的关系。

一、数据输入

假设数据放在了文本文件 c:\work\class.txt 中，没有列标题，各变量上下对齐。我们先把数据读入一个 S 数据框对象中：

```
> cl <- read.table("c:/work/class.txt",
+   col.names=c("Name", "Sex", "Age", "Height", "Weight"),
+   row.names="Name")
> cl
        Sex Age Height Weight
Alice     F  13   56.5   84.0
Becka     F  13   65.3   98.0
Gail      F  14   64.3   90.0
Karen     F  12   56.3   77.0
Kathy     F  12   59.8   84.5
Mary      F  15   66.5  112.0
Sandy     F  11   51.3   50.5
Sharon    F  15   62.5  112.5
Tammy     F  14   62.8  102.5
Alfred    M  14   69.0  112.5
Duke      M  14   63.5  102.5
Guido     M  15   67.0  133.0
James     M  12   57.3   83.0
Jeffrey   M  13   62.5   84.0
John      M  12   59.0   99.5
Philip    M  16   72.0  150.0
Robert    M  12   64.8  128.0
Thomas    M  11   57.5   85.0

William   M  15   66.5  112.0
```

二、探索性数据分析（EDA）

首先我们先研究各变量的分布情况，看分布是否接近正态，有无明显的异常值，有没有明显的序列相关，等等。

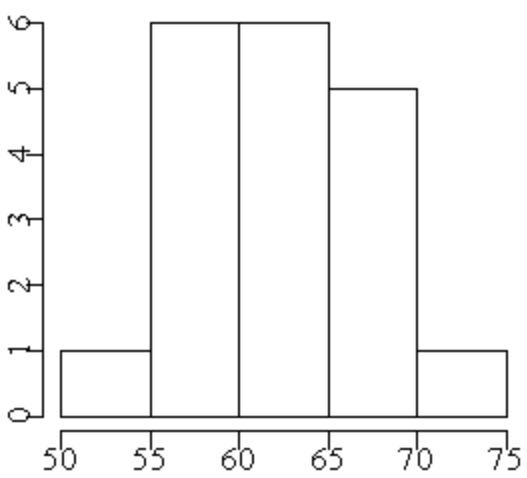
研究连续型变量的分布，可以使用直方图、盒形图、分布密度估计图和正态概率图。研究离散型变量分布只要画其分布频数条形图即可，分布频数用 table 函数计算。

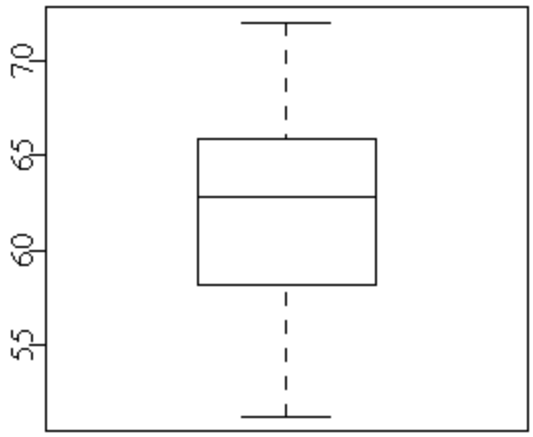
研究序列相关性可以作时间序列图和自相关函数图。因为这些图经常重复使用，我们把它们定义为函数，在同一页面画出：

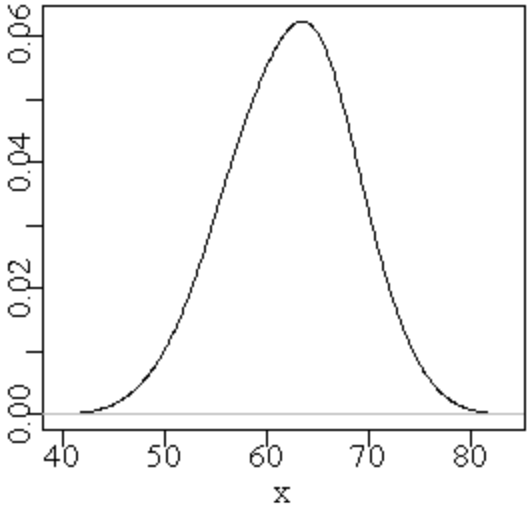
```
function(x) {
  oldpar <- par(mfrow = c(2, 2),
                mar=c(2,2,0.2,0.2),
                mgp=c(1.2,0.2,0))
  hist(x, main="", xlab="", ylab="")
  boxplot(x)
  iqd <- summary(x)[5] - summary(x)[2]
  plot(density(x,width=2*iqd), xlab = "x",
  ylab = "", type = "l", main="")
  qqnorm(x, main="", xlab="", ylab="")
  qqline(x)
  par(oldpar)
  invisible()
}

function(x) {
  oldpar <- par(mfrow=c(2,1),
                mar=c(2,2,1,0.2), mgp=c(1.2, 0.2, 0))
  plot.ts(x, main="", xlab="")
  acf(x, main="", xlab="")
  par(oldpar)
  invisible()
}
```

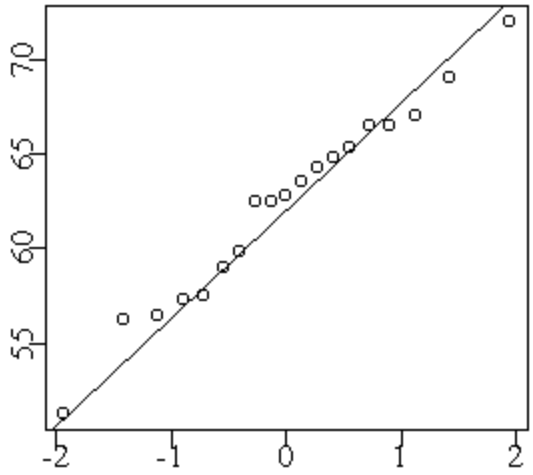
函数中最后的 invisible()表示在命令行调用此函数时不要显示任何返回值。变量 iqd 计算的是函数的四分位间距。函数 density 用来作核密度曲线估计，其 width 参数为核估计的参数。现在调用这些函数来研究各数值型变量的分布情况。在调用前先把数据框 cl 连接入当前的搜索路径中以直接使用 cl 中的变量名：

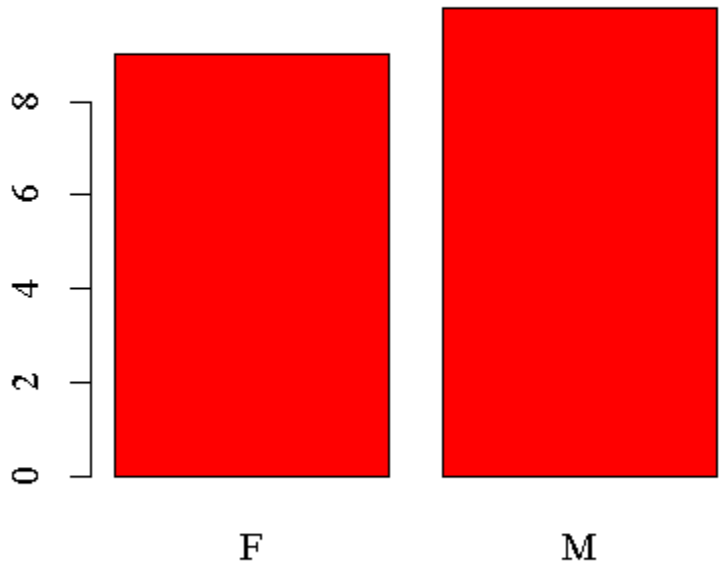
```
> attach(cl)
> eda.shape(Age)
> eda.shape(Height)
> eda.shape(Weight)
> tab.sex <- table(Sex)
> barplot(tab.sex)
```


0
1
2
3
4
5
6
50
55
60
65
70
75


55
60
65
70


0.00
0.02
0.04
0.06
40
50
60
70
80
x


55
60
65
70
-2
-1
0
1
2

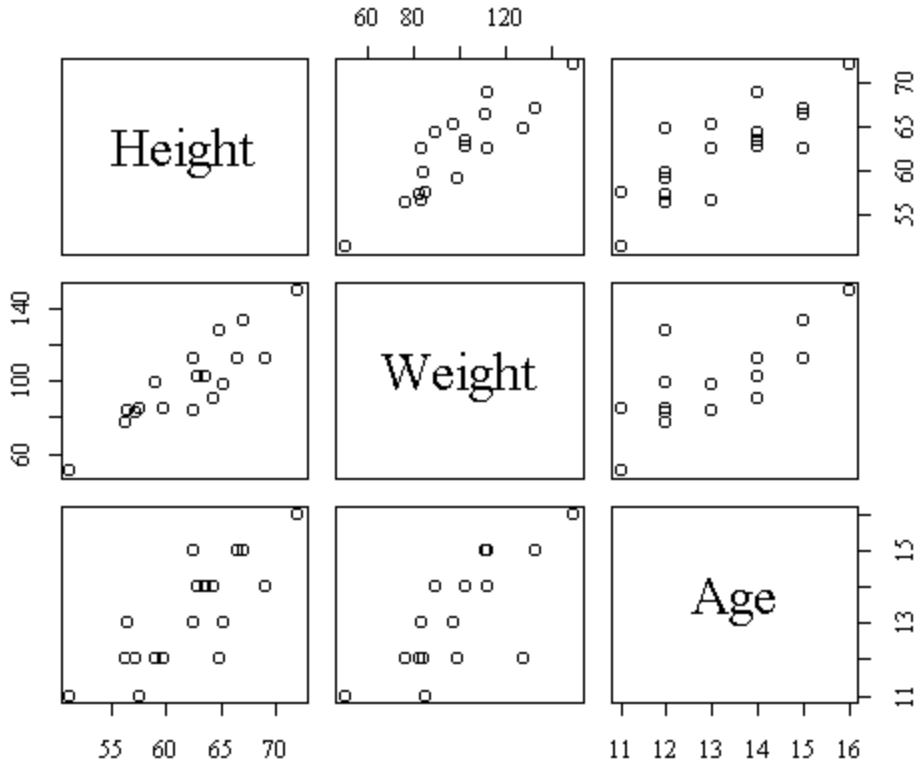


因为数据是不同个体的观测所以不可能有序列相关，未画时间序列图。这里给出了身高的分布图（图 12）及性别的频数直方图。可以看出，身高和体重都相当接近正态且无明显的异常点，体重因为取离散值所以直方图不接近正态，但从核密度估计曲线看仍可作为正态处理。

要计算一些简单统计量，可以用 summary()函数。

为了研究数值型变量 Weight、Height、Age 间的关系，我们画它们的散点图矩阵：

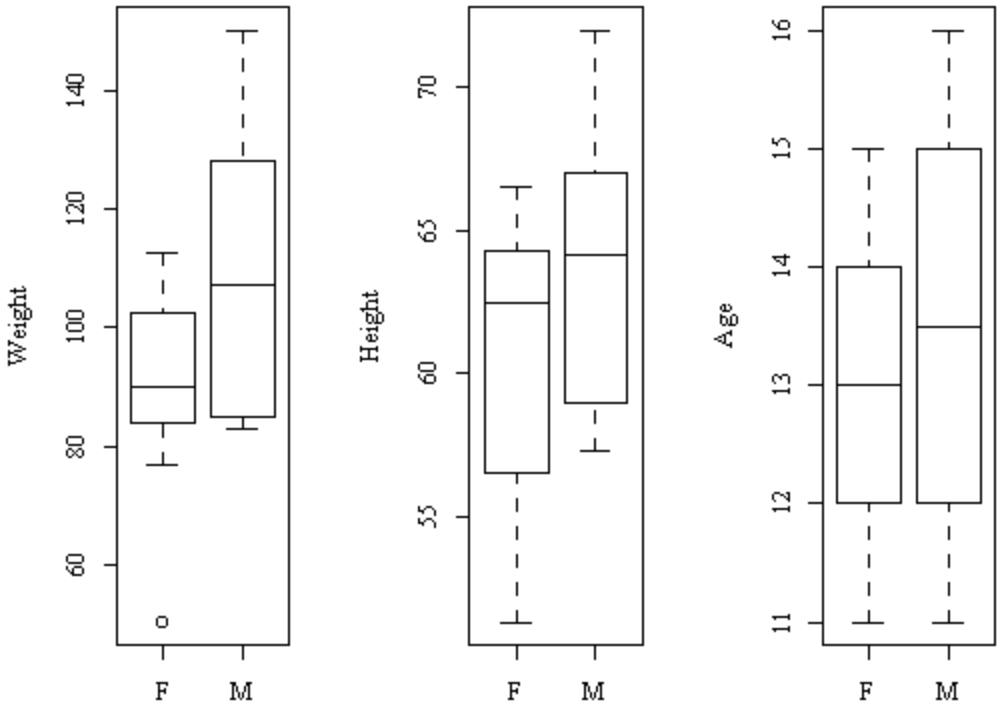
> pairs(cbind(Height, Weight, Age))



从散点图矩阵（图 14）可以看出三个变量之间都可能有线性相关关系。

为了研究因子 Sex 对其它变量的影响，可以画 Sex 不同水平上各变量的盒形图，如：

```
> par(mfrow=c(1,3))
> boxplot(Weight ~ Sex, ylab="Weight")
> boxplot(Height ~ Sex, ylab="Height")
> boxplot(Age ~ Sex, ylab="Age")
```

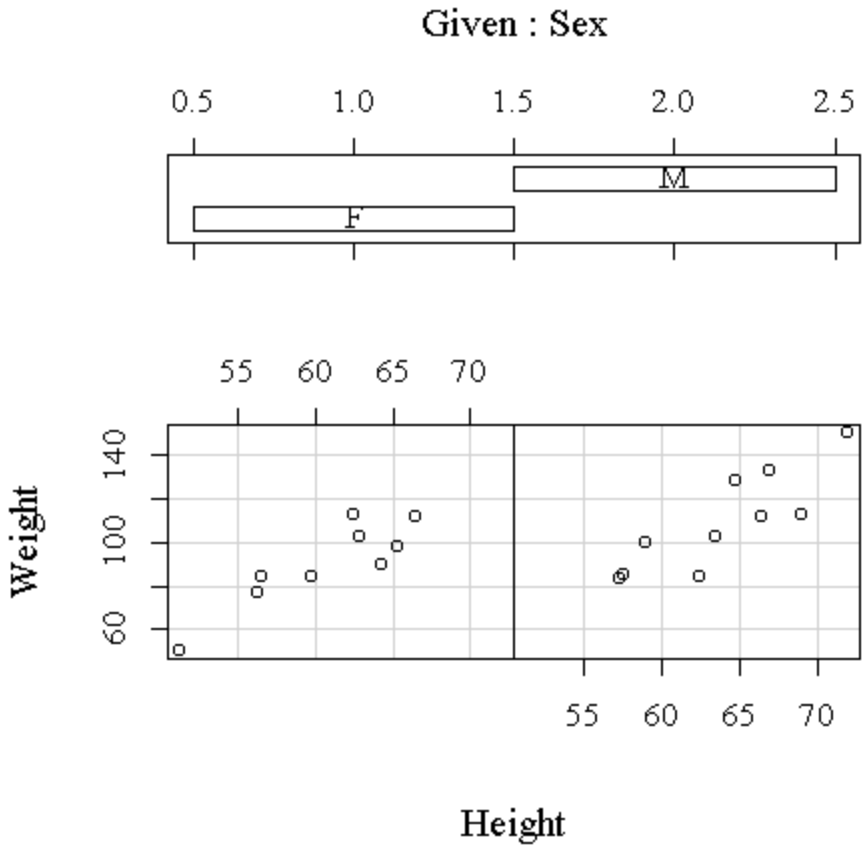


从图 15 可以看出，男女的体重、身高有明显的差别，而年龄则差别不明显。我们也可以分不同性别对某一变量分别作图或计算，这里只要使用向 Weight[Sex=="F"]，Weight[Sex=="M"] 这样的取子集的办法就可以把观测分组。更进一步还可以用函数 tapply 直接按一个因子对观测分组然后作用某个函数：

> tapply(Weight, Sex, hist)

为了研究因子 Sex 的不同水平对其它变量间的相关关系的影响，可以作协同图：

> coplot(Weight ~ Height | Sex)



结果图 16 没有反映明显的差别。

三、组间比较

我们来分析男女的身高有无显著差异，这是两组比较的问题。上面 EDA 部分的并排盒形图已经提示男女身高有明显差异，这里我们用统计假设检验给出统计结论。

男女两组可以认为是独立的，而且每组内的观测也可以认为是相互独立的。根据 EDA 结果可以认为两组都来自正态总体。这样，我们可以使用两样本 t 检验。因为方差是否相等为止，我们干脆用不要求方差相等的近似两样本 t 检验：

```
> t.test(Height[Sex="F"], Height[Sex="M"])

         Welch Two Sample t-test

data:  Height[Sex == "F"] and Height[Sex == "M"]
t = -1.4513, df = 16.727, p-value = 0.1652
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -8.155098  1.512875
sample estimates:
mean of x mean of y
```

```
60.58889  63.91000
```

结果 p 值为 0.1652，按我们一般采用的 0.05 水平是不显著的。所以从这组样本看男女的身高没有发现显著差异。

t.test 也可以进行方差相等的两组比较，以及成对比较，单总体的均值检验，详见随机文档。

类似可以进行男女体重的比较，p 值为 0.06799，也不显著。

四、回归分析

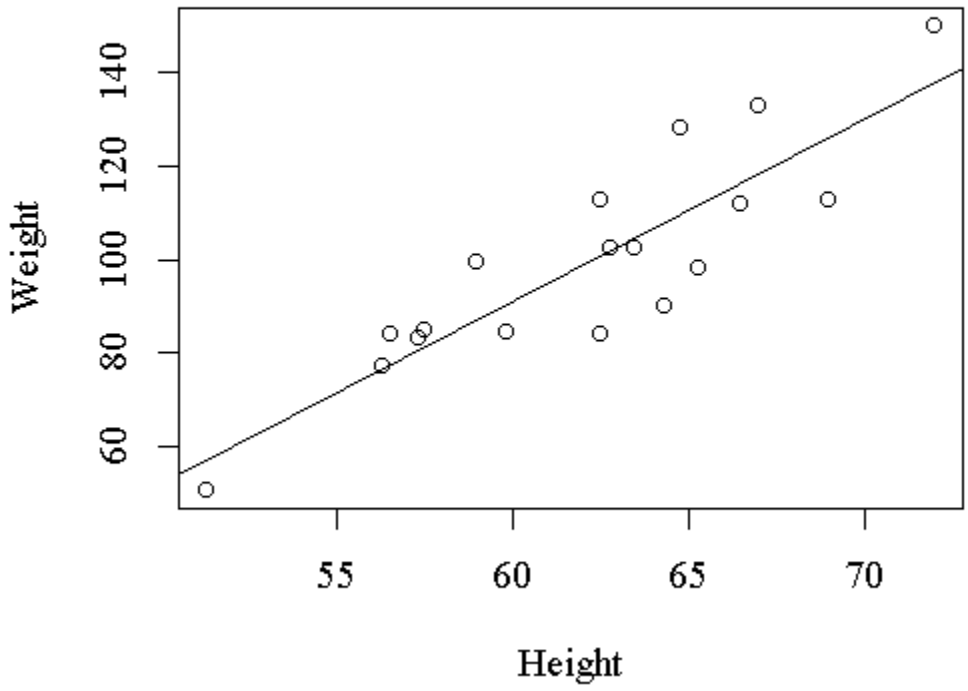
下面我们研究对体重的预报。从散点图矩阵看，体重与身高之间有明显的线性相关，所以我们先拟合一个体重对身高的一元线性回归模型：

```
> lm.fit1 <- lm(Weight ~ Height, data=cl)
> lm.fit1

Call:
lm(formula = Weight ~ Height, data = cl)

Coefficients:
(Intercept)       Height
   -143.027        3.899

> summary(lm.fit1)

Call:
lm(formula = Weight ~ Height, data = cl)

Residuals:
     Min       1Q   Median       3Q      Max
-17.6807  -6.0642   0.5115   9.2846  18.3698

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -143.0269    32.2746  -4.432 0.000366 ***
Height         3.8990     0.5161   7.555 7.89e-07 ***
---
Signif. codes:  0  `***'  0.001  `**'  0.01  `*'  0.05  `.'  0.1  ` '
1

Residual standard error: 11.23 on 17 degrees of freedom
Multiple R-Squared: 0.7705,     Adjusted R-squared: 0.757
F-statistic: 57.08 on 1 and 17 degrees of freedom,      p-value:
7.887e-007
```

拟合的模型方程为 Weight = -143.0269 + 3.8990×Height，复相关系数平方为 0.7705，检验模型的斜率为 0 的 p 值为 7.887e-007，可见模型是显著的。对于一元回归，我们可以在因变量对自变量的散点图上叠加回归直线来看回归拟合的效果（图 17）：

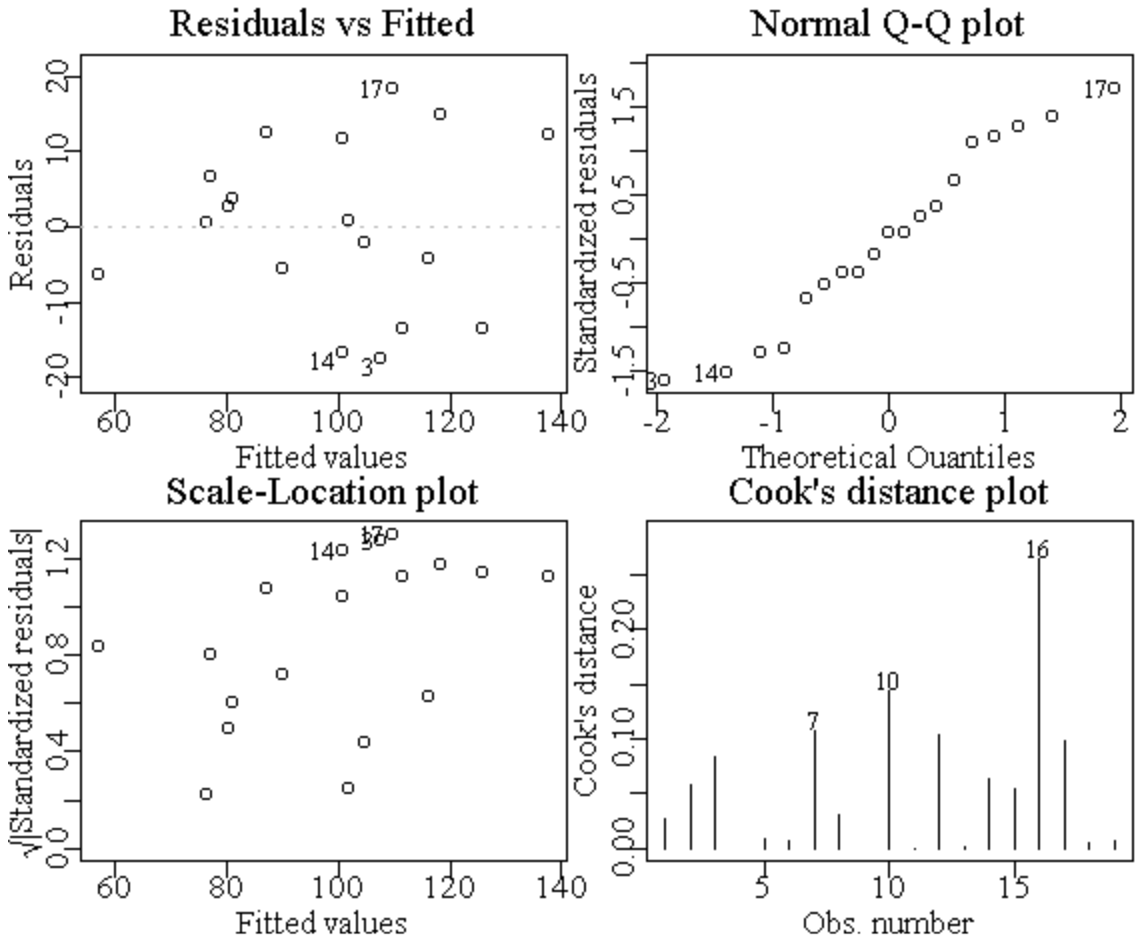
> plot(Weight ~ Height)

> abline(lm.fit1)



一般地，lm 拟合结果对象的 plot()函数可以作出若干张检查拟合效果的图形，R 可以作四个图：残差对拟合值图、残差的正态概率图、标准化残差对拟合值图、Cook 距离图。

```
> oldpar <- par(mfrow=c(2,2), mar=c(2,2,1.5,0.2), mgp=c(1.2, 0.2, 0))
> plot(lm.fit1)
> par(oldpar)
```



见图 18。如果想每个图都用全窗口来看则不要设置图形参数。四个图中，残差对拟合值图可以反映残差中残留的结构，如果模型充分的话残差应该是随机变 换没有任何模式的。残差的正态概率图可以检验线性回归假设检验的重要假定——误差项服从正态分布是否合理，可以看出残差的分布重尾、轻尾、左偏、右偏等情 况。标准化残差平方根对拟合值图可以发现残差的异常值点，即拟合最差的点。Cook 距离衡量每一观测对拟合结果的影响大小，数值大的为强影响点。图中自动 标出了最突出的点。

从 lm.fit1 的回归诊断图看残差没有明显的模式，但残差分布有轻尾倾向。没有明显的异常值点。

下面我们看加入其它变量能否进一步改善模型的预报能力。用 add1 函数可以判断加入新的变量可以改善模型：

```
> add1(lm.fit1, ~ . + Age + Sex)
Single term additions

Model:
Weight ~ Height
```

```
       Df Sum of Sq     RSS     AIC
<none>              2142.49   93.78
Age     1     22.39 2120.10   95.58
Sex     1    184.71 1957.77   94.07
>
```

add1 的结果显示一个方差分析表，列出各行中<none>一行为不加变量的情况，Age 一行为加入一个变量 Age 后的情况，Sex 一行为加入一个变量 Sex 的情况。各列中 DF 为此变量的自由度，Sum of Sq 为该变量对应的平方和，RSS 为加入该变量后的残差平方和，AIC 为加入该变量后的 AIC 统计量值。AIC 较小的模型为较好的，所以如果加入某个变量后的 AIC 减小就可以加入此变量。这里加入 Age 和加入 Sex 都使 AIC 变大，所以不应加入这两个变量。

如果一开始就加入了所有变量，可以用 drop1()函数考察去掉一个变量后 AIC 是否可以变小：

```
> lm.fit2 <- lm(Weight ~ Height + Age + Sex, data=cl)
> summary(lm.fit2)

Call:
lm(formula = Weight ~ Height + Age + Sex, data = cl)

Residuals:
     Min       1Q   Median       3Q      Max
-19.6540  -6.5737   0.4602   7.6708  20.8515

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -125.1151    33.9038  -3.690  0.00218 **
Height         2.8729     0.9971   2.881  0.01142 *
Age            3.1131     3.2362   0.962  0.35132
SexM           8.7443     5.8350   1.499  0.15472
---
Signif. codes:  0  `***'  0.001  `**'  0.01  `*'  0.05  `.'  0.1  ` '
1

Residual standard error: 11.09 on 15 degrees of freedom
Multiple R-Squared: 0.8025,     Adjusted R-squared: 0.763
F-statistic: 20.31 on 3 and 15 degrees of freedom,      p-value:
1.536e-005

> drop1(lm.fit2)
Single term deletions

Model:
Weight ~ Height + Age + Sex
       Df Sum of Sq     RSS     AIC
<none>              1844.01   94.93
Height  1   1020.61 2864.62  101.30
Age     1    113.76 1957.77   94.07
Sex     1    276.09 2120.10   95.58
>
```

从 summary()的结果看 Age 和 Sex 就不显著。用 drop1()发现去掉 Age 可以使 AIC 从 94.93 变小为 94.07，所以应该去掉 Age。对去掉 Age 后的模型再用 drop1()发现 Sex 也应该去掉。所以我们最后得到的模型时 lm.fit1。

在修改模型或数据改变后重新拟合时还可以使用 update 函数，比如要从 lm.fit2 中去掉 Age 就可以用：

lm.fit3 <- update(lm.fit2, . ~ . - Age)

在自变量个数较多时 S 提供了一个 step()函数用来进行逐步回归，它从一个初始模型开始自动判断增加或去掉变量，最后得到较好的模型：

```
> lm.step <- step(lm.fit0, ~ . + Height + Age + Sex)
Start:  AIC= 119.75
 Weight ~ 1

         Df Sum of Sq    RSS    AIC
+ Height  1    7193.2 2142.5   93.8
+ Age     1    5124.5 4211.2  106.6
+ Sex     1    1681.1 7654.6  118.0
<none>                9335.7  119.7

Step:  AIC= 93.78
 Weight ~ Height

         Df Sum of Sq    RSS    AIC
<none>                2142.5   93.8
+ Sex     1     184.7 1957.8   94.1
+ Age     1      22.4 2120.1   95.6
- Height  1    7193.2 9335.7  119.7
>
```

得到适当的模型以后，我们可以用模型进行拟合或预报。拟合只要对模型拟合结果用 predict() 函数，如:

```
> predict(lm.fit1)
    Alice     Becka      Gail     Karen     Kathy      Mary     Sandy
 77.26829 111.57976 107.68073  76.48849  90.13509 116.25859  56.99333
   Sharon     Tammy    Alfred      Duke     Guido     James   Jeffrey
100.66247 101.83218 126.00617 104.56150 118.20811  80.38752 100.66247
     John    Philip    Robert    Thomas   William
 87.01587 137.70326 109.63024  81.16732 116.25859
>
```

对新数据如果想作预报，可以在对 predict 加入新数据的数据框作为参数：

```
> new.data <- data.frame(Height=c(50, 51.2, 68, 69.7))
> predict(lm.fit1, new.data)
        1         2         3         4
 51.92459  56.60343 122.10714 128.73549
```

用 S 作随机模拟计算

作为统计工作者，我们除了可以用 S 迅速实现新的统计方法，还可以用 S 进行随机模拟。随机模拟可以验证我们的算法、比较不同算法的的优缺点、发现改进统计方法的方向，是进行统计研究的最有力的计算工具之一。

随机模拟最基本的需要是产生伪随机数，S 中已提供了大多数常用分布的伪随机数函数，可以返回一个伪随机数序列向量。这些伪随机数函数以字母 r 开头， 比如 rnorm()是正态伪随机数函数，runif()是均匀分布伪随机数函数，其第一个自变量是伪随机数序列长度 n。关于这些函数可以参见第 14 节以及 系统帮助文件。下例产生 1000 个标准正态伪随机数：

```
> y <- rnorm(1000)
```

这些伪随机数函数也可以指定与分布有关的参数，比如下例产生 1000 个均值为 150、标准差为 100 的正态伪随机数：

```
> y <- rnorm(1000, mean=150, sd=100)
```

产生伪随机数序列是不重复的，实际上，S 在产生伪随机数时从一个种子出发，不断迭代更新种子，所以产生若干随机数后内部的随机数种子就已经改变了。 有时我们需要模拟结果是可重复的，这只要我们保存当前的随机数种子，然后在每次产生伪随机数序列之前把随机数种子置为保存值即可：

```
> the.seed <- .Random.seed
> ……
> .Random.seed <- the.seed
>  y <- rnorm(1000)
```

作为例子，我们来产生服从一个简单的线性回归的数据。

```
# 简单线性回归的模拟
lm.simu <- function(n){
    # 先生成自变量。假设自变量 x 的取值范围在 150 到 180 之间，大致服从正态分布。
    x <- rnorm(n, mean=165, sd=7.5)
    # 再生成模型误差。假设服从 N(0, 1.2)分布
    eps <- rnorm(n, 0, 1.2)
    # 用模型生成因变量
    y <- 0.8 * x + eps
    return(data.frame(y,x))
}
```

S 没有提供多元随机变量的模拟程序，这里给出一个进行三元正态随机变量模拟的例子。假设要三元正态随机向量 $X \sim N(\mu, \Sigma)$ 的 *n* 个独立观测，可以先产生 n 个服从三元标准正态分布的观测，放在一个 *n* 行 3 列的矩阵中：

U <- matrix(rnorm(3*n), ncol=3, byrow=T)

可以认为矩阵 U 的每一行是一个标准的三元正态分布的观测。设矩阵 Σ 的 Choleski 分解为 $\Sigma = A'A$， *A* 为上三角矩阵，若随机向量 $\xi \sim N(0, I)$，则 $\mu + A'\xi \sim N(\mu, \Sigma)$。因此， $\mu + A'U'$ 作为一个三行 *n* 列的矩阵每一行都是服从 $N(\mu, \Sigma)$ 分布的，且各行之间独立。经过转置，产生的 *X*

X <- matrix(rep(mu,n), ncol=3, byrow=T) + U %*% A

是一个 *n* 行三列的矩阵。

有时模拟需要的计算量很大，多的时候甚至要计算几天的时间。对于这种问题我们要善于把问题拆分成可以单独计算的小问题，然后单独计算每个小问题，把结果保存在 S 对象中或文本文件中，最后综合保存的结果得到最终结果。

如果某一个问题需要的计算时间比较长，我们在编程时可以采用以下的技巧：每隔一定时间就显示一下任务的进度，以免计算已经出错或进入死循环还不知 道；应该把中间结果每隔一段时间就记录到一个文本文件中（cat()函数可以带一个 file 参数和 append 参数，对这种记录方法提供了支持），如果需 要中断程序，中间结果可能是有用的，有些情况下还可以根据记录的中间结果从程序中断的地方继续执行。

S 常用函数参考

这一节分类列出常用的函数，需要时可以参看帮助。

基本

一、数据管理

```
vector：向量              numeric：数值型向量              logical：逻辑型向量
character；字符型向量          list：列表              data.frame：数据框
c：连接为向量或列表            length：求长度    subset：求子集
seq，from:to，sequence：等差序列
rep：重复         NA：缺失值              NULL：空对象
sort，order，unique，rev：排序
```

unlist：展平列表
attr，attributes：对象属性
mode，typeof：对象存储模式与类型
names：对象的名字属性

二、字符串处理

character：字符型向量　　nchar：字符数　　substr：取子串
format，formatC：把对象用格式转换为字符串
paste，strsplit：连接或拆分
charmatch，pmatch：字符串匹配
grep，sub，gsub：模式匹配与替换

三、复数

complex，Re，Im，Mod，Arg，Conj：复数函数

四、因子

factor：因子　　codes：因子的编码 levels：因子的各水平的名字
nlevels：因子的水平个数　　cut：把数值型对象分区间转换为因子
table：交叉频数表　　split：按因子分组
aggregate：计算各数据子集的概括统计量
tapply：对“不规则”数组应用函数

数学

一、计算

+，-，*，/，^，%%，%/%：四则运算
ceiling，floor，round，signif，trunc，zapsmall：舍入
max，min，pmax，pmin：最大最小值
range：最大值和最小值
sum，prod：向量元素和，积
cumsum，cumprod，cummax，cummin：累加、累乘
sort：排序
approx 和 approx fun：插值
diff：差分
sign：符号函数

二、数学函数

abs，sqrt：绝对值，平方根
log，exp，log10，log2：对数与指数函数
sin，cos，tan，asin，acos，atan，atan2：三角函数
sinh，cosh，tanh，asinh，acosh，atanh：双曲函数

beta，lbeta，gamma，lgamma，digamma，trigamma，tetragamma，pentagamma，choose，lchoose：与贝塔函数、伽玛函数、组合数有关的特殊函数

fft，mvfft，convolve：富利叶变换及卷积
polyroot：多项式求根
poly：正交多项式
spline，splinefun：样条差值
besselI，besselK，besselJ，besselY，gammaCody：Bessel 函数
deriv：简单表达式的符号微分或算法微分

三、数组

array：建立数组　　matrix：生成矩阵
data.matrix：把数据框转换为数值型矩阵
lower.tri：矩阵的下三角部分　　mat.or.vec：生成矩阵或向量
t：矩阵转置　　cbind：把列合并为矩阵　　rbind：把行合并为矩阵
diag：矩阵对角元素向量或生成对角矩阵
aperm：数组转置　　nrow，ncol：计算数组的行数和列数
dim：对象的维向量 dimnames：对象的维名
row/colnames：行名或列名　　%*%：矩阵乘法
crossprod：矩阵交叉乘积（内积）　　outer：数组外积
kronecker：数组的 Kronecker 积　　apply：对数组的某些维应用函数
tapply：对“不规则”数组应用函数　　sweep：计算数组的概括统计量
aggregate：计算数据子集的概括统计量　　scale：矩阵标准化
matplot：对矩阵各列绘图　　cor：相关阵或协差阵
Contrast：对照矩阵　　row：矩阵的行下标集
col：求列下标集

四、线性代数

solve：解线性方程组或求逆　　eigen：矩阵的特征值分解
svd：矩阵的奇异值分解　　backsolve：解上三角或下三角方程组
chol：Choleski 分解　　qr：矩阵的 QR 分解
chol2inv：由 Choleski 分解求逆

五、逻辑运算

<，>，<=，>=，==，!=：比较运算符
!，&，&&，|，||，xor()：逻辑运算符
logical：生成逻辑向量　　all，any：逻辑向量都为真或存在真
ifelse()：二者择一　　match，%in%：查找
unique：找出互不相同的元素　　which：找到真值下标集合
duplicated：找到重复元素

六、优化及求根

optimize，uniroot，polyroot：一维优化与求根

程序设计

一、控制结构

if，else，ifelse，switch：分支
for，while，repeat，break，next：循环
apply，lapply，sapply，tapply，sweep：替代循环的函数。

二、函数

function：函数定义　　source：调用文件　　call：函数调用
.C，.Fortran：调用 C 或者 Fortran 子程序的动态链接库。
Recall：递归调用
browser，debug，trace，traceback：程序调试
options：指定系统参数　　missing：判断虚参是否有对应实参
nargs：参数个数　　stop：终止函数执行
on.exit：指定退出时执行　　eval，expression：表达式计算
system.time：表达式计算计时　　invisible：使变量不显示
menu：选择菜单（字符列表菜单）

其它与函数有关的还有：delay，delete.response，deparse，do.call，dput，environment ，，formals，format.info，interactive，is.finite，is.function，is.language，is.recursive ，match.arg，match.call，match.fun，model.extract，name，parse，substitute，sys.parent ，warning，machine。

三、输入输出

cat，print：显示对象
sink：输出转向到指定文件
dump，save，dput，write：输出对象
scan，read.table，load，dget：读入

四、工作环境

ls，objects：显示对象列表　　rm，remove：删除对象
q，quit：退出系统　　.First，.Last：初始运行函数与退出运行函数。
options：系统选项　　?，help，help.start，
apropos：帮助功能
data：列出数据集

统计计算

一、统计分布

每一种分布有四个函数：d——density（密度函数），p——分布函数，q——分位数函数，r——随机数函数。比如，正态分布的这四个函数为 dnorm，pnorm，qnorm，rnorm。下面我们列出各分布后缀，前面加前缀 d、p、q 或 r 就构成函数名：

norm：正态，t：t 分布，f：F 分布，chisq：卡方（包括非中心）
unif：均匀，exp：指数，weibull：威布尔，gamma：伽玛，beta：贝塔
lnorm：对数正态，logis：逻辑分布，cauchy：柯西，
binom：二项分布，geom：几何分布，hyper：超几何，nbinom：负二项，pois：泊松
signrank：符号秩，wilcox：秩和，tukey：学生化极差

二、简单统计量

sum, mean, var, sd, min, max, range, median, IQR（四分位间距）等为统计量，sort，order，rank 与排序有关，其它还有 ave，fivenum，mad，quantile，stem 等。

三、统计检验

R 中已实现的有 chisq.test，prop.test，t.test。

四、多元分析

cor，cov.wt，var：协方差阵及相关阵计算
biplot，biplot.princomp：多元数据 biplot 图
cancor：典则相关　　princomp：主成分分析
hclust：谱系聚类　　kmeans：k-均值聚类
cmdscale：经典多维标度
其它有 dist，mahalanobis，cov.rob。

五、时间序列

ts：时间序列对象　diff：计算差分　time：时间序列的采样时间　window：时间窗

六、统计模型

lm，glm，aov：线性模型、广义线性模型、方差分析

关于在 R 中使用 C 程序的一些问题

R 是一个优秀的统计计算语言，但是因为它是解释型语言，所以在对数组元素的迭代运算方面会很慢。在 R 用 C 语言程序可以既保留 R 的易用性又可以在必要时提高速度。本文讲述在 Windows 环境下如何用 Borland 的 C 编译器来完成 R 和 C 的结合。

假设我们要用 C 编码的问题是两个向量的卷积问题，当然，R 中已经有 convolve 可以实现，我们这里只是作为一个例子。两个无穷向量 x 和 y 的卷积定义为 z[i] = sum_{k} x[i-k]*y[k]，两个有限长度向量的卷积是把向量两端填零后卷积然后取非填零部分计算得到的结果。 C 程序为：

```
void convolve(double *a, int *na, double *b, int *nb, double *ab){
    int i, j, nab = *na + *nb - 1;
    for(i = 0; i < nab; i++)
    ab[i] = 0.0;
    for(i = 0; i < *na; i++)
    for(j = 0; j < *nb; j++)
    ab[i + j] += a[i] * b[j];
}
```

这里所有参数都使用指针，是因为 R 中基本的运算单元是向量，即 C 中的数组（指针）。

一、如何用 Borland 自由 C++编译器生成 Win32 环境下的 DLL(无图形界面成分)

首先要获得 Borland 的自由编译器，网址为 http://www.borland.com/bcppbuilder/freecompiler/ 假设安装在 c:\borland\bcc55 中。在 bin 子目录中，生成一个 bcc32.cfg 文件，包含如下行：

```
-I"c:\Borland\Bcc55\include"
-L"c:\Borland\Bcc55\lib"
```

生成一个 ilink32.cfg，包含如下行:

```
-L"c:\Borland\Bcc55\lib"
```

生成如下的批处理文件 mkdll.bat:

```
REM Make DLL for R using borland free compiler 5.5
PATH=c:\borland\bcc55\bin;C:\;c:\windows;c:\windows\command
bcc32 -O2 -WDE %1
```

假设我们有一个 C 程序文件 testdll.c，其中函数 convolve 是我们希望输出的，我们需要另外生成一个文件 testdll.def，其中包含如下行:

```
LIBRARY testdll
```

```
EXPORTS
 _convolve
```

其中函数名前的下划线是编译器加的。如果有多个输出函数可以写在后面，每个函数名占一行。

然后，在 DOS 窗口中运行

```
mkdll testdll.c
```

即可得到所需的 testdll.dll 库。注意 mkdll.bat 文件应该在路径中，否则应该使用全路径。

二、如何用 BorlandCBuilder 生成 Win32 环境下的 DLL(无图形界面成分)

从文件菜单选择 new...，然后选择 Console wizard，选 Window type 为 console，Execution type 为 EXE，即可生成程序框架，把要输出到 DLL 中的函数写到源程序文件中，并且要输出的函数用

```
extern  "C"  __declspec(dllexport) __stdcall
```

定义，如

```
extern  "C"  __declspec(dllexport) __stdcall
void convolve(double *a, int *na, double *b, int *nb, double *ab){
```

。编译后可以得到所需的 DLL 库。

三、如何为 R 的 Windows 版准备用.C 界面调用 C 或 C++的动态链接库

用如上方法得到得到 DLL 库 testdll.dll 后，进入 R，运行

```
    dyn.load("testdll.dll")
```

可以把 DLL 库调入，注意如果库不在当前工作路径可以使用全路径，如"c:/work/testdll.dll"。

然后，在 R 中定义如下包装函数 conv：

```
conv <- function(a, b){
    .C("_convolve", as.double(a), as.integer(length(a)),
       as.double(b), as.integer(length(b)),
       ab=double(length(a)+length(b)-1))$ab
}
```

运行如：

```
    > conv(c(1,2), c(10,20,30))
    [1] 10 40 70 60
```

注意：

（1） R 中所有对象传递到 C 中都是数组（即指针），所以在我们的 C 程序中虽然参量 na 是一个整数标量我们还是用指针表示。

（2) R 函数".C"的第一参数是 C 中的函数名，用 Borland 5.5 编译时会自动加上前导下划线，其它参数都必须用 as.double，as.int 等包裹起来，否则无法与 C 中的类型对应。".C"返回一个列表，其中包含 每一个调用参数（除函数名外），有名调用的参数结果列表中元素名就使用给定的名字。如本例中".C"函数返回一个包含元素"$ab"和四个无名元素的列 表。

（3) 如果修改了 C 程序，要重新调入 DLL 库，可以先用 dyn.unload("test.dll") 卸载然后再用 dyn.load 调入。

四、如何为 R 的 Windows 版准备用.Call 界面调用 C 或 C++的动态链接库

用.C 界面可以很容易地从 R 中调用一个字符终端格式地 Windows DLL，但是不能实现从 C 中直接使用 R 对象的功能。如果想从 C 中直接使用 R 对象，就需要使用 R 的.Call 界面来调用动态链接入的函数，而在 C 程序中，使用 SEXP 来说明 R 对象，并嵌入头文件 R.h 和 Rdefines.h。具体操作步骤如下：

（1) 安装 Borland 自由 C++编译器并设置好。

（2) 获得 R 的源程序包并安装在适当位置，如 C:\R-1.2.1。

（3) 安装 R Windows 运行版本于适当位置，如 C:\rw1021。

（4) 进入 DOS 命令窗口，确保程序搜索路径中有 Borland 编译器的路径， 进入 C:\rw1021\bin，运行

```
IMPDEF R.DEF R.DLL
```

这样可以得到 R.DLL 库中所有输出函数列表于 R.DEF 中。 然后，运行

```
IMPLIB R.LIB R.DEF
```

这样可以得到用来与 R.DLL 链接的库 R.LIB。

（5) 生成一个用来制作 DLL 库的批命令文件 mkrdll.bat，内容为

```
REM Make DLL calling R using borland free compiler 5.5
PATH=c:\borland\bcc55\bin;C:\;c:\windows;c:\windows\command
bcc32 -u -O2 -WDE -IC:\R-1.2.1\src\include -LC:\rw1021\bin %1 R.lib
```

（6) 以上步骤只需要做一次。对 C 的源文件 call.c，只要运行 mkrdll call.c 就可以生成所需得到 DLL 库 call.dll。以卷积为例，C 源程序如下：

```
#include
#include

SEXP convolve2(SEXP a, SEXP b)
{
```

```
    int i, j, na, nb, nab;
    double *xa, *xb, *xab;
    SEXP ab;
    PROTECT(a = AS_NUMERIC(a));
    PROTECT(b = AS_NUMERIC(b));
    na = LENGTH(a); nb = LENGTH(b); nab = na + nb - 1;
    PROTECT(ab = NEW_NUMERIC(nab));
    xa = NUMERIC_POINTER(a); xb = NUMERIC_POINTER(b);
    xab = NUMERIC_POINTER(ab);
    for(i = 0; i < nab; i++) xab[i] = 0.0;
    for(i = 0; i < na; i++)
    for(j = 0; j < nb; j++) xab[i + j] += xa[i] * xb[j];
    UNPROTECT(3);
    return(ab);
}
```

（7) 在 R 中，用

```
    > dyn.load("call.dll")
```

调入 DLL 库，然后用如

```
    > .Call("_convolve", c(1,2), c(10,20,30))
    [1] 10 40 70 60
```

用 R 内建函数计算为

```
    > convolve(c(1,2), rev(c(10,20,30)), type="open")
    [1] 10 40 70 60
```

注：

（1) 关于如何在 C 中调用 R 对象及函数的问题详见 R 随机手册 “Writing R Extensions”的第 4 章。

（2) 从上面的例子中我们可以看到， 为了能够从 C 中使用 R 对象，需要调入头函数 R.h 和 Rdefines.h。 所有 R 对象都要说明成 SEXP 类型。

（3) 可以用 AS_NUMERIC(a)把 R 对象 a 强制转换为数值向量类型的 R 对象， 以确保输入参数类型是正确的，但这一步最好是在 R 包裹代码中完成。 用 NEW_NUMERIC(nab)可以生成一个新的 R 数值向量对象，长度 nab。

（4) 所有在 C 代码中生成或重新赋值的 R 对象如上面的 a,b,ab 在赋值时都要用 PROTECT 宏保护起来，格式如 PROTECT(ab = NEW_NUMERIC(nab))。 在推出函数之前要用 UNPROTEC 函数去掉对新对象的保护， 参数为保护的对象的个数。

（5) 为了得到 R 数值向量的元素，用 xa=NUMERIC_POINTER(a)可以得到该向量的头指针， 这是一个普通 C 的 double 类型指针，我们就可以把 xa 当作一个 C 数组处理了。

（6) 函数可以返回一个 R 对象，即返回值类型为 SEXP。

练习

.写出元素为 _的向量。

.写出从 .开始每次增加 .，长度为 .的向量。

.写出.重复 .次的向量。

.对向量 .，写出其元素大于等于 .小于 .的条件。

.对向量 .，写出其元素都等于 .的条件。

.写出包含 .个月份名称的向量。

.写出包含方程 $z^6 = 1$ 的根的向量，并写出其幅角的余弦和正弦值。

.设 .为一个长 .的整数向量。比如，__。

.显示 .第 .到 .号元素。

.把 .第 .，.，.号元素赋值为 .。

.显示 .中除了第 .号和第 .号的元素之外的子集。

.列出 .中个位数等于 .的元素。

.列出 .中个位数等于 .的元素的下标位置。

.给 .的每一个元素加上名字，为 .到 .。

.求 .的平均值并求每一个元素减去平均值后的离差，计算 .的离差平方和及元素的平方和。

.把 .从大到小排序。计算 .的 .分位数到 .分位数之间的距离。

.定义一个维数为.的数组其第一层（第三下标为 .）取从 .开始的奇数，第二层取从 .开始的偶数。显示每一层的第 .行元素。把第.，_号元素赋值为零。把第一层加上 .，把第二层加上 .。分别计算第一层和第二层的平均值。

.对线性模型 $Y = X\beta + \varepsilon$，写出当 .满秩时计算 β 的 .表达式。写出估计 ε 的方差的 . 表达式。

. 把 ▁数据中的性别、年龄、身高分别输入到 .中。计算不同性别、不同年龄的人数，并计算每一组的平均身高。把这些变量组合成一个列表。把 ▁数据输入为 .的数据框。

. 把语句 ▁所生成的向量保存到一个文本文件中，数据项用空格和换行分隔。从此文件中读入数据到向量 .中。

. .设 .是一个长度为 .的向量，写一段程序，计算 .的长度为 .的滑动和：

$$S_x(t) = \sum_{i=0}^{s-1} x_{t-i}, \quad t = s, s+1, \ldots, n$$

. 写一个 . 的模拟函数：

$$x_t = a + bx_{t-1} + \varepsilon_t, t = 1, 2, \ldots, n, Var(\varepsilon_t) = \sigma^2$$

函数的参数为 .、 .、 .、 x_0 和 σ，缺省时 .，.，.，x_0 .，$\sigma =$.。