

写在前面

经过一星期的工作，我终于完成了《S-Plus 使用简介》。S-Plus 是 S 语言的运行环境，用 S 语言进行统计分析具有诸多优点，可以很方便地完成几乎所有的统计分析工作，因此，如果专门从事统计分析工作，最好能掌握它。

本电子书将重点放在了 S 语言的介绍上，S 语言是一个较大的语言系统，S-Plus2000 所带的 PDF 文件共有上千页，本电子书只是对 S 语言简单扼要的介绍，若要详细了解 S 语言，可参照 S-Plus2000 中的 PDF 文件。

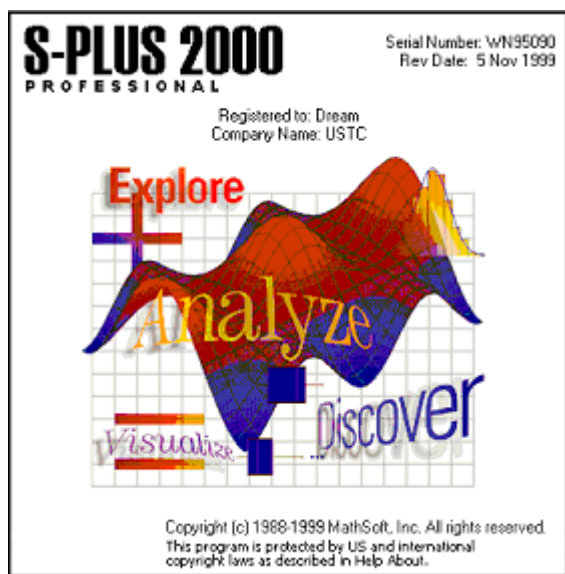
本电子书是为参加美国大学生数学建模竞赛而编的参考资料，但依我所见，若只是进行简单的数据分析，完全没有必要专门学习 S 语言，Matlab 中有一个统计工具箱，其中提供了大量的统计函数，用其便可以完成几乎所有简单的统计功能，诸如统计作图、参数估计、假设检验、回归分析、方差分析等等，大家可以参考科学出版社的《Matlab 数理统计(6.x)》。

本电子书参考并拷贝了大量电子资料，在此表示感谢。

中国科大 孟强

2003. 1. 25

第一章 S-Plus 简介



S-Plus 是由美国 MathSoft 公司开发的一种基于 S 语言的统计学软件，是世界上公认的三大统计软件之一，主要用于数据挖掘、统计分析和统计作图等等。S-Plus 的最大特点在于它可以交互地从各方面发现数据中的信息，并可以很容易地实现一个新的统计方法。另外，S-Plus 的数据可以直接的来源于 Excel, Lotus, Access, SAS, SPSS 等软件，其兼容性极好。

本章中，我们简单地熟悉一下 S-Plus 的桌面环境，从下一章起，我们将系统地介绍 S 语言。

§ 1.1 S-Plus 的桌面环境

进入 S-Plus 系统，便直接进入了 Commands 窗口，Commands 窗口由提示符“>”开始，后面便可以直接写 S 语言代码，由于 S 语言是一种解释性语言，因此，输入代码敲回车后，便可直接在 Commands 窗口看到运行结果，其运行方式和 Matlab 相似，在此不再赘述。

下面是一个 S 语言的程序实例，其具体的语法含义和函数功能我们将在以后介绍。

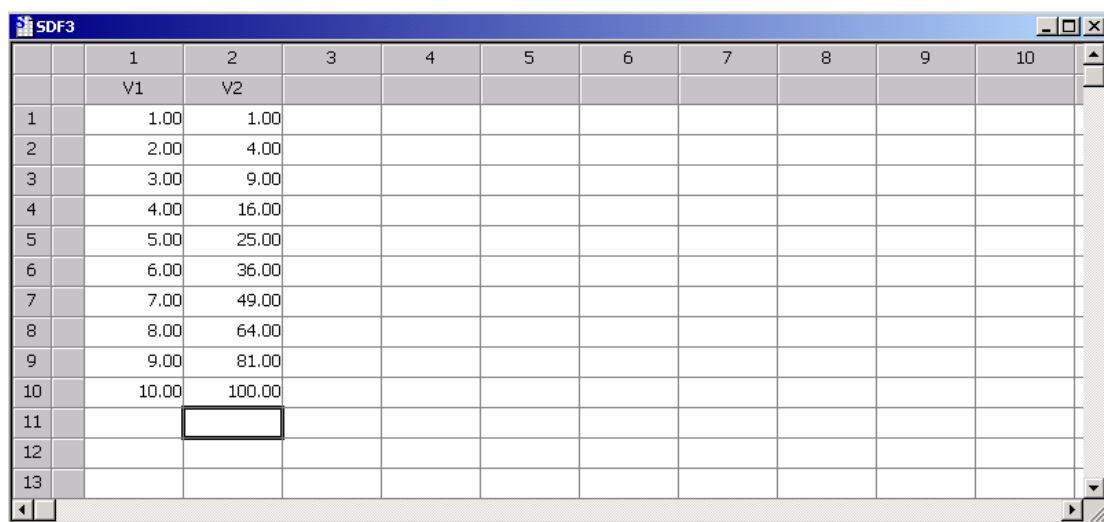
> marks <- c(10, 6, 4, 7, 8)	将几个数字组成一个列向量
> mean(marks)	求该向量的算术平均值
[1] 7	得出结果
> median(marks)	求该向量的中位数
[1] 7	得出结果
> min(marks)	求向量元素的最小值

[1] 4	得出结果
> max(marks)	求向量元素的最大值
[1] 10	得出结果
> boxplot(marks)	绘出盒形图

§ 1.2 使用 S_Plus 绘制数据图形

得到一批数据，我们最常做的工作是绘出其数据图，比如柱状图，饼状图，拟合曲线等等，S_Plus 中有专门的图形工具来实现其功能。

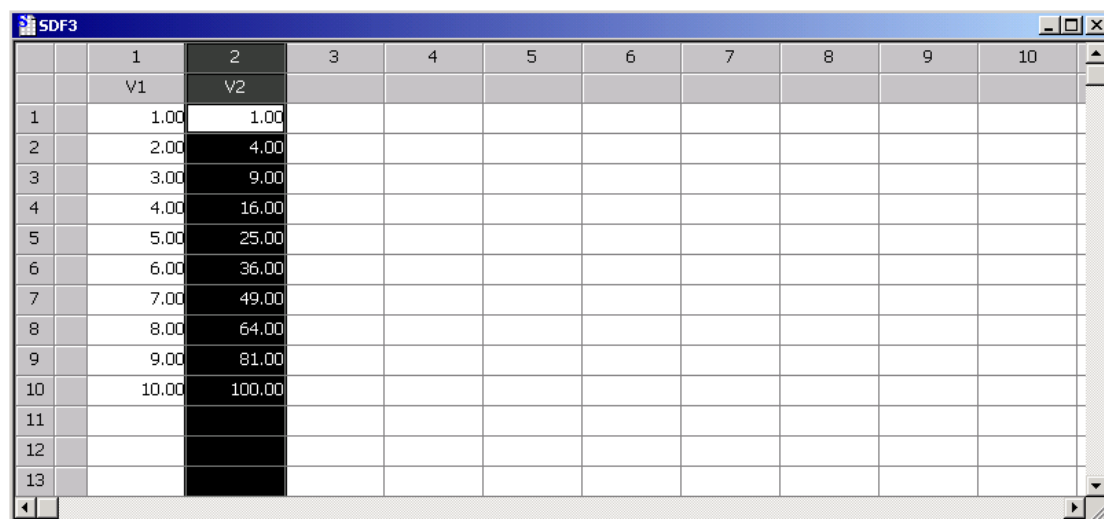
首先，我们建立一个数据文件，在菜单中点击 **File→new**，便出现一个选择窗口，选择 Data Set，便出现图 1.2.1，在其中输入数据即可。



	1	2	3	4	5	6	7	8	9	10
	V1	V2								
1	1.00	1.00								
2	2.00	4.00								
3	3.00	9.00								
4	4.00	16.00								
5	5.00	25.00								
6	6.00	36.00								
7	7.00	49.00								
8	8.00	64.00								
9	9.00	81.00								
10	10.00	100.00								
11										
12										
13										


图 1.2.1

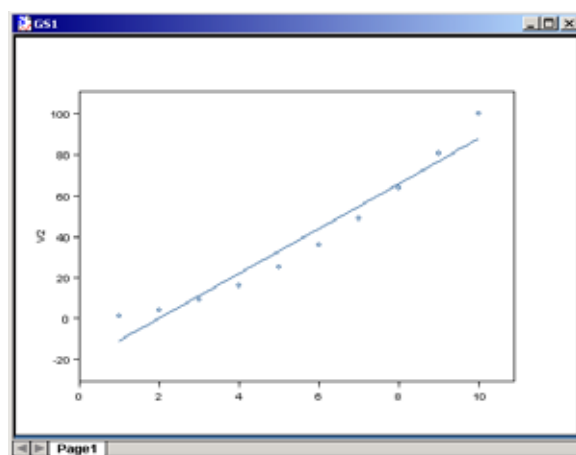
选中所要画图的一列数据，如图 1.2.2



	1	2	3	4	5	6	7	8	9	10
	V1	V2								
1	1.00	1.00								
2	2.00	4.00								
3	3.00	9.00								
4	4.00	16.00								
5	5.00	25.00								
6	6.00	36.00								
7	7.00	49.00								
8	8.00	64.00								
9	9.00	81.00								
10	10.00	100.00								
11										
12										
13										

图 1.2.2

点击菜单栏中的 ，便出现下图中的选择窗口，选择所需的图形类型，便可立即绘出相应的图形。



数据的直线拟合

第二章 S 向量

§ 2.1 S 语言简介

S_Plus 基于 S 语言，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。

从本章起到第十三章，我们来介绍 S 语言。本章介绍 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、数组、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。

§ 2.2 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）。

§ 2.3 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 中标量常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变量，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。

§ 2.4 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 $p \times n$ 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

§ 2.5 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=”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）。

§ 2.6 逻辑向量

向量可以取逻辑值，如：

```
>1<-c(T, T, F)
>1
[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
```

```
>1<-x>3
```

```
>1
```

```
[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。

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
```

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 必须把TRUE 转换为数值型的1，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”

§ 2.7 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'。

§ 2.8 复数向量

S 支持复数运算。复数常量只要用我们通常的如`3.5+2.1i` 的格式即可。`complex` 模式的向量为复数元素的向量。可以用`complex()`函数生成复数向量（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。

§ 2.9 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
[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, 例如:

```
>x [-(1:2)]
[1] 6.25
```

表示扣除相应的元素。

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
```

可见`x[x<10]`取出所有小于10 的元素组成的子集。这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]
```

第三章 S对象

S是一种基于对象的语言。S 的对象包含了若干个元素作为其数据，另外还可以有一些

特殊数据称为属性（attribute），并规定了一些特定操作（如打印、绘图）。比如，一个向量是一个对象，一个图形是一个对象。S 并不是一个象C++那样的面向对象语言，它的数据具有一些对象的特点，并不支持类机制。

S 对象分为单纯的（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；复合对象的元素可以是不同类型的对象，每一个元素是一个对象。向量（vector）是单纯对象。

§ 3.1 固有属性：mode和length

S 对象都有两个基本的属性（attribute）：mode（类型）属性和length（长度）属性。比如向量的类型为logical（逻辑型）、numeric（数值型）、complex（复数型）、character（字符型），比如mode(c(1,3,5)>5)结果为字符串“logical”。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）。后面讲到的列表类型为list。

要判断某对象是否某类型，有许多个类似于is.numeric()的函数可以完成。is.numeric(x)用来检验对象x 是否数值型，它返回一个逻辑型结果。is.character()可以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，比如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。

§ 3.2 修改对象长度

对象可以取0 长度或正整数为长度。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？只要给它赋一个长度短的子集就可以了。例如：

```
>x<-1:3
>x<-1:4
>x<-x [1:2]
>x
[1] 1 2
```

§ 3.3 attributes()和attr()函数

attributes(object)返回对象object 的各特殊属性组成的列表，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, "name")
[1]"apple""orange"
```

也可以把`attr()`函数写作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"
```

§ 3.4 对象的class属性

在S 中可以用特殊的`class` 属性来支持面向对象的编程风格，对象的`class` 属性用来区分对象的类，可以写出通用函数根据对象类的不同进行不同的操作，比如，`print()`函数对于向量和矩阵的显示方法就不同，`plot()`函数对不同类的自变量作不同的图形。

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。

第四章 多维数组和矩阵

§ 4.1 数组和矩阵

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 。

向量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（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），列表的每个变量为一维的名字。

例如上面的`z`可以这样定义：

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

函数`matrix()`用来定义数组中的一种最常用的：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，列表第一个变量是长度与行数相等的字符型向量，表示每行的标签，列表第二个变量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。

§ 4.2 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的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`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。

还有一种特殊下标办法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表达式看作是对数组的数据向量取子集。

§ 4.3 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，其方法是使用一个二维数组作为数组的下标，二维数组的每一行是一个元素的下标，列数为数组的维数。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
```

§ 4.4 数组四则运算

可以对数组之间进行四则运算（+，-，*，/，^），这时进行的是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、向量的运算除外）。

§ 4.5 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）。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' 。这里因为前者较可能所以软件中表示前者，但内积最好还是用`crossprod(x)`来计算。要表示，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 时即计算 X 自身的内积 $X'X$ 。

其它运算还有`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。详见随机文档，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)`返回有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。

§ 4.6 矩阵合并与拉直

函数`cbind()`把其自变量作为子矩阵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，自变量如果是子矩阵其高度（行数）应该相等，`rbind` 的自变量是矩阵或看作行向量的向量，自变量如果是矩阵其宽度（列数）应该相等。如果参加合并的自变量是矩阵且比其它自变量则可以循环使用。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矩阵（用`cbind(x)`）或 $1 \times n \times 1$ 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。

§ 4.7 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
```

§ 4.8 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 $z = \frac{\cos y}{1+x^2}$ 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 $0, 1, \dots, 9$

中取值。假设四个元素 a, b, c, d 都是离散均匀分布的相互独立的随机变量，我们设法求矩阵行列式 $ad-bc$ 的分布。首先，随机变量 ad, bc 和同分布，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，元素名为取值，而元素值为频数，比如 `fr[1]` 的元素名为 -81 ，值为 19，表示值 -81 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。

§ 4.9 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 `b` 把 `a` 的第 2 维移到了第 1 维，`a` 的第 3 维移到了第 2 维，`a` 的第 1 维移到了第三维。这时有 `b[i,j,k]=a[j,k,i]`。

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。

§ 4.10 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，如果我们要把矩阵a 的各列分别排序，只要用apply(a, 2, sort)，这样对每一列排序得到一个向量用第一维来引用，恰好得到所需结果；但是，如果要行排序，则apply(a, 1, sort)把a 的每一行排序后的结果用第一维来引用，就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：

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

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。

第五章 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"))
```



```
>table(sex)
```

```
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。

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。

第六章 列表

§ 6.1 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，这里暂不讨论。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）。

§ 6.2 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**”，这时缺失值的记号。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，没有任何特殊意义。）

§ 6.3 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** 方便的一个地方。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。

第七章 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、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。

§ 7.1 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`）。

§ 7.2 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"
```

§ 7.3 attach()函数

数据框的主要用途是保存统计建模的数据。`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。

第八章 输入输出

§ 8.1 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 =", i, "\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。

§ 8.2 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。

第九章 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。

§ 9.1 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
```

其中的表达式也可以是用大括号包围的复合表达式。例如:

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

可以在变量`lambda` 缺失时赋一个值。又比如

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

考虑一个向量`x`，打印它元素的值和重对数值，但这只有在元素都为正且对数都为正时才能做到。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。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，其第一参数是一个表达式，根据此表达式的值决定返回第几个参数的值。第一参数结果为数值型时取对应位置的参数，结果为字符型时返回由此字符串指定名字的参数。如果字符型表达式结果不能匹配，则取最后一个无名参数的值，不存在无名参数时返回NULL。例如:

```
>x<-2
>switch(x,"one","two","three")
[1]"two"
>x<-"Jan"
>switch(x,Jan="Still cold",Feb="Hopeful","Not considered")
[1]"Still cold"
>x<-"Mar"
>switch(x,Jan="Still cold",Feb="Hopeful","Not considered")
[1]"Not considered"
```

§ 9.2 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
```

从这个例子我们也可以看出，显式的循环经常是可以避免的，利用函数可以对每个元素计算值、sum 等统计函数、apply、lapply、sapply、tapply 等可以代替循环。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。

`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。

第十章 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。这样作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。

§ 10.1 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" "gl" "marks" "ns"  
[10] "pl" "rec" "tmp.x" "x" "x1" "x2" "x3" "y"  
>rm(x, x1, x2, x3)  
>ls()  
[1] "A" "Ai" "b" "cl" "cl.f" "fit1" "gl" "marks" "ns"  
[10] "pl" "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。

§ 10.2 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。

§ 10.3 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。实参先与指定了虚参名的结合，其它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。

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
```

§ 10.4 作用域

函数的虚参在函数体内是局部变量，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。

§ 10.5 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，可以在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` 中用`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。

第十一章 S统计模型

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。

§ 11.1 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。

`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：

```
add() 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
<code>anova(对象1, 对象2)</code>	把一个子模型与原模型比较，生成方差分析表。
<code>coefficients(对象)</code>	返回回归系数（矩阵）。可简写为 <code>coef(对象)</code> 。
<code>deviance(对象)</code>	返回残差平方和，如有权重则加权。
<code>formula(对象)</code>	返回模型公式。
<code>plot(对象)</code>	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。
<code>predict(对象, newdata=数据框)</code>	
<code>predict.gam(对象, newdata=数据框)</code>	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）。
	<code>predict.gam()</code> 与 <code>predict()</code> 作用相同但适用性更广，可应用于 <code>lm</code> 、 <code>glm</code> 和 <code>gam</code> 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 <code>predict.gam()</code> 函数可以避免由此引起的偏差。
<code>print(对象)</code>	简单显示模型拟合结果。一般不用 <code>print()</code> 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。

§ 11.2 方差分析

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。

第十二章 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 U 的Choleski分解为 $U = A \Sigma A^T$ ， A 为上三角矩阵，若随机向量 ϵ ，则 $X = \mu + U\epsilon$ 。因此， X 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。

第十三章 S常用函数参考

这一章分类列出常用的函数，需要时可以参看帮助。

§ 13.1 基本函数

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

§ 13.2 数学函数

计算类:

+, -, *, /, ^, %%, %/%: 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

数学函数类:

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

数组类:

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

线性代数类:

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

逻辑运算类:

<, >, <=, >=, ==, !=: 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

优化及求根类:

optimize, uniroot : 一维优化与求根