R程序结构

理工大学理学院–周世祥

2015年10月27日

目录

1 R程序基本流程

1.1 内置函数

R里面有很多内置函数，这些内置函数保存在不同的扩展包里。

- 有些内置函数保存在R的核心包里，比如mean()函数保存在base包里；

- 有些内置函数分散在各个扩展包里，比如做k-近邻分类方法的knn()保存在class包里；

R里有非常丰富的内置函数，大大方便了我们的数据分析工作。内置函数的使用比较简单，穿插在各个部分讲解，本章重点讲解如何编写自己的函数。在编写自己的函数之前，先介绍一下R的条件控制语句和循环语句。

1.2 条件控制语句

1.2.1 if/else语句

if/else语句是分支语句中主要的语句，if/else语句的格式为：

"if(cond) statement_1 "

"if(cond) statement_1 else statement_2 "

即如果条件cond成立，则执行"statement_1"；否则跳过。第二句是指如果条件cond成立，则执行"statement_1"；否则执行"statement_2"。

更复杂的情况：

```
  if (cond_1)
statement_1
   else if (cond_2)
statement_2
   else if (cond_3)
      statement_3
   else
      statement_4
```

例如：

```
x<-3
if(x>2) y=2*x else y=3*x
#假如x大于2，则返回2*x，否则返回3*x
 y

## [1] 6
```

```
x<-1.5
if(x>2) y=2*x else y=3*x
#假如x大于2，则返回2*x，否则返回3*x
y

## [1] 4.5
```

1.2.2 ifelse语句

ifelse 结构是if/else紧凑的向量化版本，其语法为

`ifelse(cond,statement1,statement2)`

如 cond 成立，则执行statement1,否则执行statement2.

例如：

```
x<-1
ifelse(x>2, y<-2*x, y<-3*x) #假如x大于2，则返回2*x，否则返回3*x

## [1] 3
```

1.2.3 switch语句

switch语句是多分支语句，其使用方法为：

`switch(statement,list)`

其中，statement是表达式，list是列表，可以用有名定义。如果表达式的返回值在1到length(list)，则返回列表相应位置的值。

例如：

```
  switch(1,2*3,sd(1:5),runif(3))
## [1] 6
 #返回（2*3,sd(1:5),runif(3)）list中的第一个成分
 switch(2,2*3,sd(1:5),runif(3))  #返回第二个成分
## [1] 1.581139
  switch(3,2*3,sd(1:5),runif(3))  #返回第三个成分
## [1] 0.4547600 0.1450915 0.2899514
```

当list是有名定义时，statement等于变量名时，返回变量名对应的值；否则，返回NULL值。例如

```
  x<-"meat"
  switch(x, meat="chicken", fruit="apple", vegetable="potato")
## [1] "chicken"
```

1.3 循环语句

有for循环、while循环和repeat循环语句。

1.3.1 for循环

” for(ind in expr_1) expr_2 ”

其中，ind是循环变量，”expr_1” 是一个向量表示式（通常是个序列，如-10:10，”expr_2” 通常是一组表达式。

例如：Fibonacci序列是数学中著名的序列，前两个元素都是1，第三个元素是第一、二元素的和，第四个元素是第二、三个元素的和，一直下去。为了生成Fibonacci前16个元素，程序：

```
Fibonacci<-NULL #生成一个空置向量
Fibonacci[1]<-Fibonacci[2]<-1 # Fibonacci向量的第1和2个元素赋值为1
n=16
 for (i in 3:n) Fibonacci[i]<-Fibonacci[i-2]+Fibonacci[i-1] #用for执行循环语句
 Fibonacci
```

```
## [1]   1   1   2   3   5   8  13  21  34  55  89 144 233 377 610 987
```

1.3.2 while循环

while (condition) expr

只有当condition条件成立的时候，执行表达式expr。

例如：编写小于1000的Fibaonacci序列：

```
Fibonacci[1]<-Fibonacci[2]<-1 # Fibonacci向量的第1和2个元素赋值为1
i<-1
while (Fibonacci[i]+Fibonacci[i+1]<1000) {
  #用while执行循环语句
      Fibonacci[i+2]<-Fibonacci[i]+Fibonacci[i+1]
      i<-i+1      }
Fibonacci
```

```
## [1]   1   1   2   3   5   8  13  21  34  55  89 144 233 377 610 987
```

1.3.3 repeat语句

repeat循环依赖break语句跳出循环。

例：利用repeat生成小于1000的Fibonacci序列

```
Fibonacci[1]<-Fibonacci[2]<-1 # Fibonacci向量的第1和2个元素赋值为1
```

```
i<-1
repeat {    #用repeat执行循环语句
  Fibonacci[i+2]<-Fibonacci[i]+Fibonacci[i+1]
  i<-i+1
  if (Fibonacci[i]+Fibonacci[i+1]>=1000) break
}
Fibonacci
```

```
##  [1]   1   1   2   3   5   8  13  21  34  55  89 144 233 377 610 987
```

此处，break为中止语句。R中同样用break语句对循环进行终止，使程序跳到循环以外。

1.4 编写自己的函数

在较复杂的计算问题中，有时候一个任务可能需要重复多次，这时我们不妨编写自己的函数。

这么做的好处之一是可以批量处理这些任务，而不需要每次都重复执行。

另一个好处是函数内的变量名是局部的，即当函数运行结束后它们不再被保存到当前的工作空间，这就可以避免许多不必要的混淆和内存空间占用。

R语言与其它统计软件最大的区别之一是你可以编写自己的函数，而且可以像使用R的内置函数一样使用你的函数。

编写函数的句法是：

```
函数名 = function (参数1, 参数2…)
{
  statements
return(object)
}
```

函数的每一部分都很重要，接下来我们逐一详细介绍。

函数名可以是任何值，但以前定义过了的要小心使用，后来定义的函数会覆盖原先定义的函数。一旦你定义了函数名，你就可以像R的其它函数

一样使用。例如：

可以这样定义一个用std求标准差的函数(R内置的求标准差的函数是sd)

```
std = function(x) { sqrt(var(x)) }
```

对于这类只有一个语句的简单函数，也可不要，即

std = function(x) sqrt(var(x))

这里的函数名就是std，以后我们就可以像其它函数一样使用它了。

```
x=c(1,3,5,7,9)
std(x)

## [1] 3.162278
```

假如我们不使用圆括号，直接输入函数名，按回车键将显示函数的定义式：

```
std

## function(x) { sqrt(var(x)) }

function(x) sqrt(var(x))

## function(x) sqrt(var(x))
```

编写函数一定要写上function这个关键词，它告诉R这个新的数据对象是函数，所以编写函数时千万不可忘记。

函数根据实际需要的不同而有不同的参数设置，下面将介绍三种情况：

1. 无参数：有时编写函数是为了某种方便，函数每次的返回值都是一样的，其输入不是那么重要。比如我们编写“welcome”函数，其每次返回值都是“welcome use R”。

```
welcome = function() print("welcome to use R")
welcome()

## [1] "welcome to use R"
```

2. 单参数：假如要使你的函数个性化，可以使用单参数，函数将会根据参数的不同，返回值也不同。

```
 welcome.sb = function(names) print(paste("welcome",names,"to use R"))
welcome.sb("Mr fang")

## [1] "welcome Mr fang to use R"
```

3. 默认参数：即不输入任何参数。上面的welcome.sb函数假如不输入参数结果将会怎么样呢？

```
# welcome.sb()
```

错误在paste(”welcome”, names, ”to use R”)：缺少变元“names”，也没有缺失值。

没有输入参数，函数welcome.sb将返回出错信息。其实我们可以给函数设置默认值，R提供了一个简单的方法允许给函数的参数设置默认值。比如：

```
welcome.sb=function(names="Mr fang")print(paste("welcome", names,"to use R"))
welcome.sb()

## [1] "welcome Mr fang to use R"
```

下面编写一个模拟函数求服从均值为10，标准差为5的正态样本数据的t统计量。

```
 sim.t=function(n){
mu=10;sigma=5;
x=rnorm(n,mu,sigma)
#生成n个均值为mu,标准差为sigma的正态分布随机数
(mean(x)-mu)/(sd(x)/sqrt(n))
}
sim.t(5)   # 样本量为5

## [1] 0.3084454
```

sim.t函数的均值、标准差都是固定的，但是假如我们希望这个函数的均值、标准差是可随意设置的。这时，我们就要在函数里添加均值、标准差两个参数。例如：

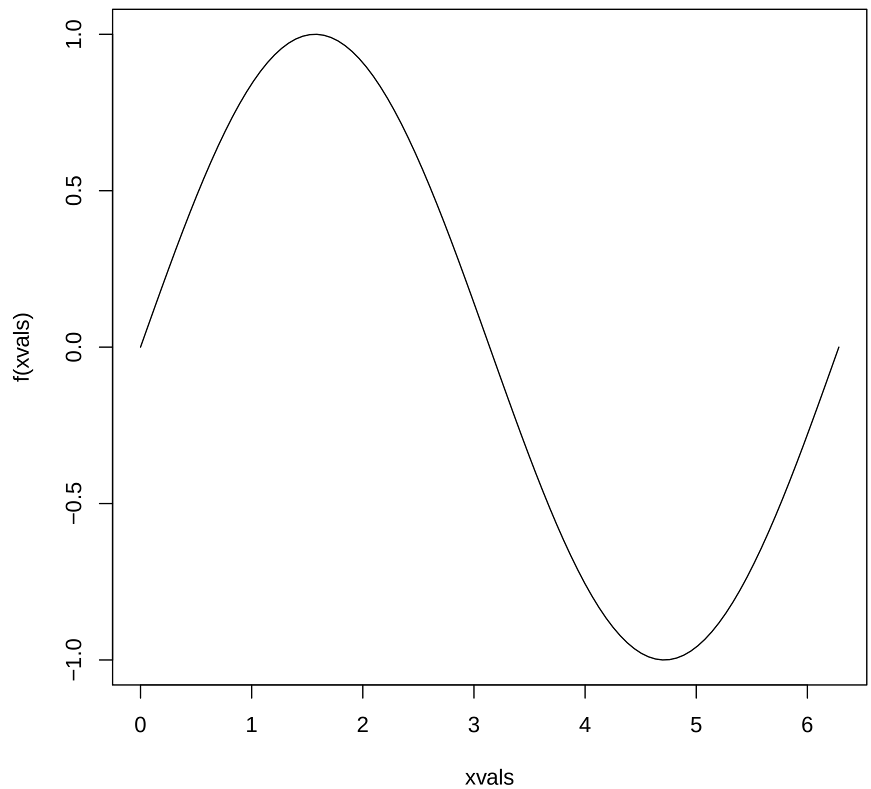
```
 sim.t = function(n,mu=10,sigma=5){
   x=rnorm(n,mu,sigma)
   (mean(x)-mu)/(sd(x)/ sqrt(n))
}
  sim.t(5)             # 样本含量为5，均值为10，标准差为5

## [1] 0.9428387

 sim.t(5,0,1)          # 样本含量为5，均值为0，标准差为1

## [1] -0.697225

  sim.t(5,4)            # 样本含量为5，均值为4，标准差为5

## [1] 0.3424085

   sim.t(5,sigma=100)   # 样本含量为5，均值为10，标准差为100

## [1] 0.5287646

  sim.t(5,sigma=100,mu=1) #样本含量为5，均值为1，标准差为100

## [1] 3.389341
```

这里值得注意的一点是，不要把位置参数与名义参数混淆起来。位置参数必须与函数定义的参数顺序一一对应，比如sim.t(5,0,1)，5对应参数n，0对应参数mu，1对应参数sigma。

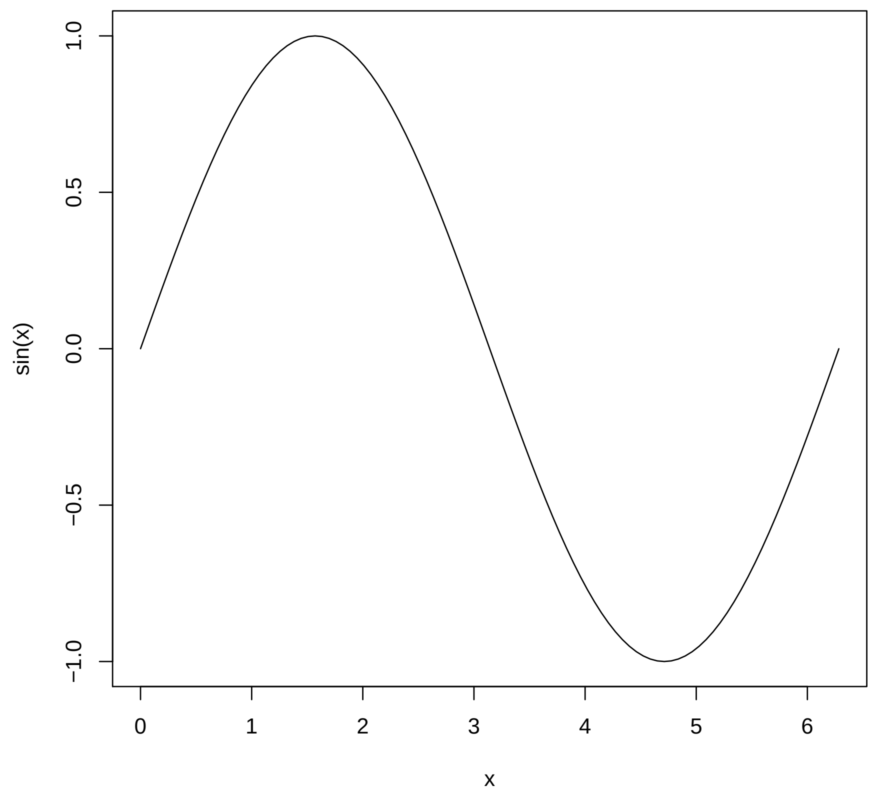
再比如sim.t(5,4)，5对应参数n，4对应参数mu，第三个位置上没有值与参数sigma对应，这时sigma取默认值5。但是使用名义参数，就没有按顺序对应，比如sim.t(5,sigma=100,mu=1)，这使多参数函数使用起来非常方便。

R语言允许定义一个变量，然后将变量值传递给R的内置函数。这在作图上非常有用。比如编写一个画图函数，允许你先定义一个变量x，用这个变量生成y变量，然后描出它们的图像。

```
 plot.f=function(f,a,b,...){
xvals=seq(a,b,length=100) #生成100个[a,b]区间内的数列
plot(xvals,f(xvals),type="l",...) }
#作横坐标为xvals，纵坐标为f(xvals)的函数图
 plot.f(sin,0,2*pi)
```



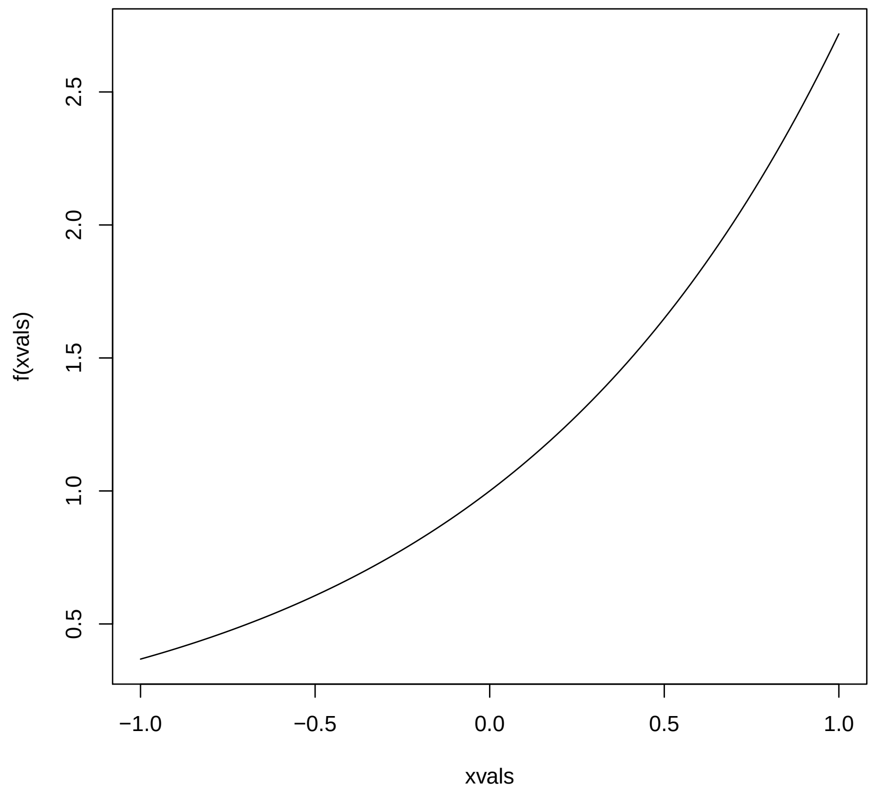

```
# 作0到2pi的正弦曲线，见图
# 这个函数的功能实际上等同于R内置函数curve。
 curve(sin,0,2*pi)
```



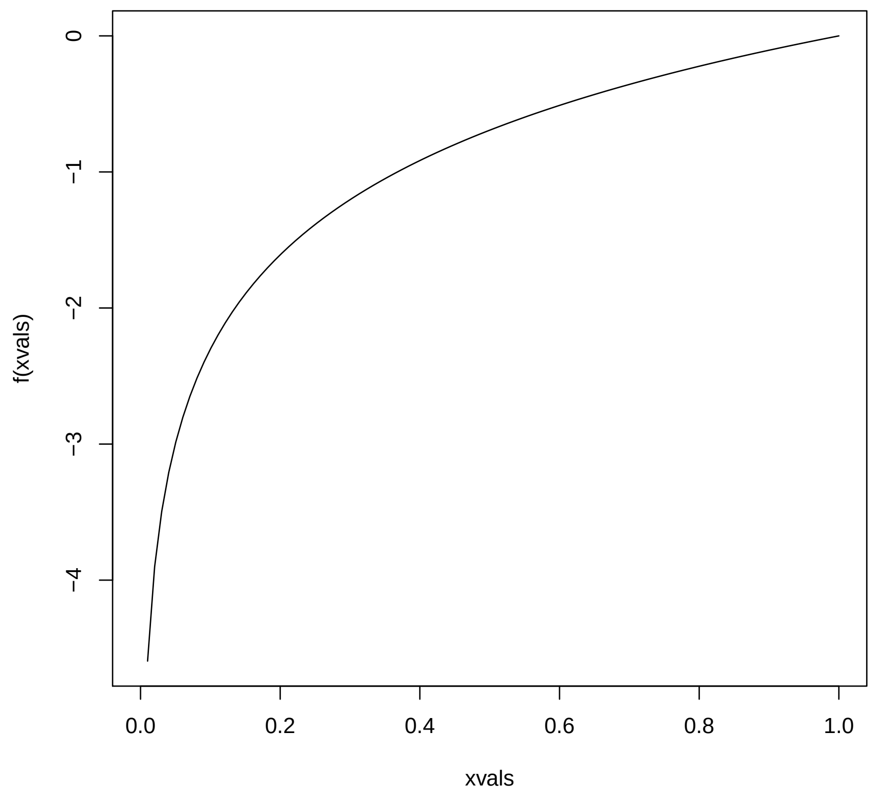

```
#作0到2pi的正弦曲线，见图
```

由plot.f建立的函数是一种泛函，f可以是任何函数，例如指数函数、对数函数，也可以自定义函数。

```
plot.f(exp,-1,1)     #作-1到1的指数曲线
```




```
plot.f(log,0,1)        #作0到1的对数曲线
```



函数体和函数返回值是整个函数的主要部分，默认返回函数体的最后一个表达式的结果。如果函数体的表达式只有一个当然就很简单。例如：

```
my.average = function(x) sum(x)/length(x)
my.average(c(1,2,3))
```

```
## [1] 2
```

当函数体的表达式超过一个时，要用{}封起来。例如我们用一个编写名为vms()的函数来计算向量的最大5个数的均值,并返回最大的5个值。R里也可以用return()返回函数需要的结果，当需要返回多个结果时，一般建议用list形式返回。

例如：

```
  vms=function(x){
xx=rev(sort(x)) #对x向量从小到大排序，然后用rev()转成从大到小排序
xx=xx[1:5] #提取前面5个元素
mean(xx) #求均值
return(list(xbar=mean(xx),top5=xx)) #返回均值和最大的5个数
}
  y=c(5,15,32,25,26,28,65,48,3,37,45,54,23,44)
  vms(y) #利用自己编写的函数vms()

## $xbar
## [1] 51.2
##
## $top5
## [1] 65 54 48 45 44
```

编写程序，尤其是编写很长的程序是一件浩大的工程，费心费力，而且随着程序写的越长，越不好管理，也不好理解。

1. 建立从上到下设计的思想，将大的程序拆分成几块来写，每一块可以写成单独的函数。这有点像是盖楼，先把桩和框架搭好，然后再往里面逐步填充内容。

2. 将每一块又分成几步来写。

3. 应及时勤快地加注释。写好的程序过了几天，往往可能忘了其含义。

4. 尽可能做向量化运算。因为R将所有对象都存储在内存中，尽量少用循环(for,while)。用R内置函数 lappply,sapply,mapply等处理向量、矩阵或列表。

5. 在完整的数据集上运行程序前，抽取部分数据子集进行测试，消除bug并进一步优化程序。

1.5 程序调试

程序调试是每个程序员都头疼但是大多情况下必须面临的问题。计算

机程序的错误或者缺陷叫“bugs”。我们所写的程序不一定都是百分百正确。调试程序一般分为如下几步：

1. 识别程序是否存在错误。有时很容易，因为如果程序出错，无法运行，那肯定是存在；但是有些时候，这类错误很难去识别。

2. 找出程序出错的原因。当程序有错，无法继续运行时，R往往会给出报错信息，可以根据报错信息找出出错的原因。在R里还提供了traceback()、Debug()、Browser()函数可以跟踪和找出程序出错的地方。

3. 修正错误并测试。

4. 寻找类似的错误

R中proc.time()函数返回当前R已经运行的时间。例如：

```
proc.time()

##    user  system elapsed
##    1.04    0.07    1.31
```

其中user是指R执行用户指令的CPU运行时间，system是指系统所需的时间，elapsed是指R打开到现在总共运行的时间。

计算程序运行时间的函数是: system.time(expr, gcFirst = TRUE)

其中，expr是需要运行的表达式，gcFirst是逻辑参数。system.time是实际上两次调用了proc.time()，在程序运行前调用一次，运行完后调用一次，然后计算两次的时间差，即为程序的运行时长。例如：

```
system.time(for(i in 1:100) mad(runif(1000)))

##    user  system elapsed
##    0.01    0.00    0.02
```

该程序等价于：

```
ptm<-proc.time() #将proc.time()的返回值保存到ptm对象里
for(i in 1:100) mad(runif(1000))
proc.time()-ptm #两者的差即运行程序所需的时间
```

```
##    user  system elapsed
##    0.01    0.00    0.02
```

R设计主要是用来基于向量和矩阵的运算。比如求两个向量的和：

程序1：

```
 ptm<-proc.time()
x<-rnorm(100000)
 y<-rnorm(100000)
 z<-c()
 for (i in 1:100000){
    z<-c(z,x[i]+y[i])
 }
 proc.time()-ptm
```

```
##    user  system elapsed
##   26.77    0.10   26.98
```

该程序写得非常没有效率，总共运用了50.41秒时间。该程序每次都增加一个元素到Z向量里，这样总共增加了100000次。如果我们既然已经知道了Z的长度，我们首先就给定z的长度，这样可以提高运行效率。

程序2

```
ptm<-proc.time()
z<-rep(NA,100000)
for (i in 1:100000){
  z[i]<-x[i]+y[i]
}
proc.time()-ptm
```

```
##    user  system elapsed
##    0.19    0.00    0.19
```

总共运行了0.51秒，比程序1效率高了很多，运行效率相当于是程序1的100倍。

程序3

```
ptm<-proc.time()
z<-x+y
proc.time()-ptm
```

```
##    user  system elapsed
##       0       0       0
```

该程序只要运行0.01秒就可以了。

2 优化问题

优化是在统计计算中非常重要的一部分内容，因为很多统计方法，比如最小二乘法和极大似然估计法，归根到底就是求目标函数的最小值或者最大值。接下来，先讲解最简单的一元函数的优化求解，然后再讲解多元函数的优化求解。

2.0.1 一元优化

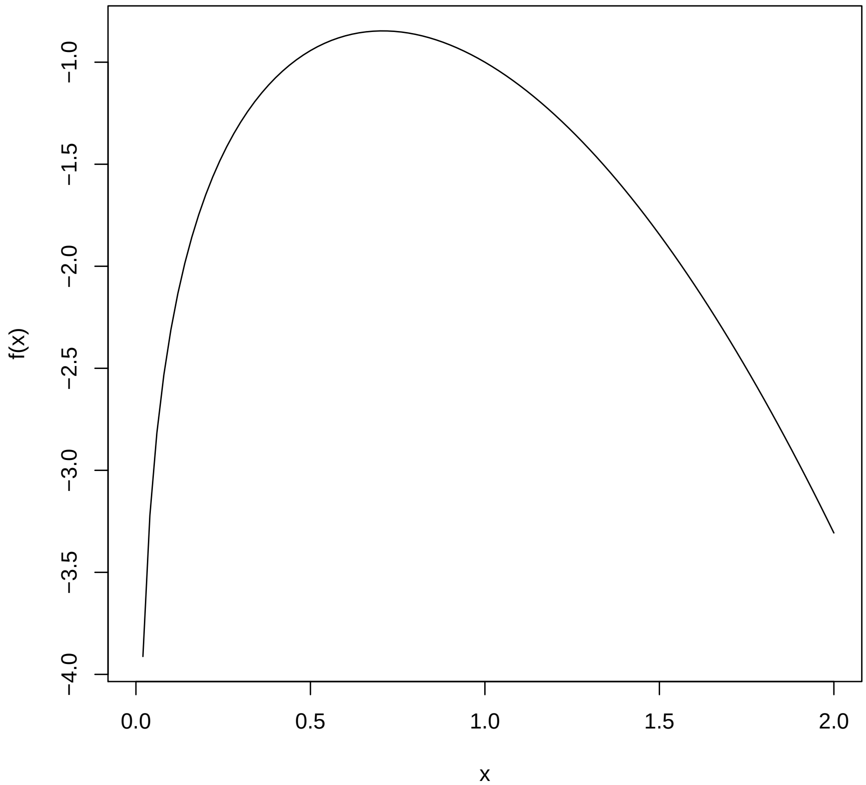
R做一元的最优化求解函数是

```
optimize(f = , interval = ,  ..., lower = min(interval),
  upper = max(interval), maximum = FALSE,
  tol = .Machine$double.eps^0.25)
```

其中，f是需要求优化的函数，interval是参数搜索的区间，lower是参数搜索的下限，如果缺失，用interval的最小值代替，upper是参数搜索的上限,缺失时用interval的最大值代替。Maximum默认是FALSE，表示默认是求极小值。tol是精度的容忍度。

例如：求解 $f(x) = \ln(x) - x^2$ 这个函数的图形如图所示。该函数只有一个极大值，利用拉格朗日方法可以求得在$x = \sqrt{2}/2$处取得最大值。

```
f=function(x) log(x)-x^2 #编写f函数
curve(f,xlim=c(0,2))     #作f函数在（0,2）区间上的图
```




```
optimize(f,c(0.1,10),tol=0.0001,maximum=T)

## $maximum
## [1] 0.7071232
##
## $objective
## [1] -0.8465736

#求f函数的最大值的返回结果
```

利用R求解出来的目标函数的最大值为-0.8465736，这与利用拉格朗日方法求解的结果相同。

2.0.2 多元函数优化

多元函数的优化求解可以用使用函数optim()求解，其用法如下

```
optim(par, fn, gr = NULL, ...,
method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN"),
lower = -Inf, upper = Inf,
control = list(), hessian = FALSE)
```

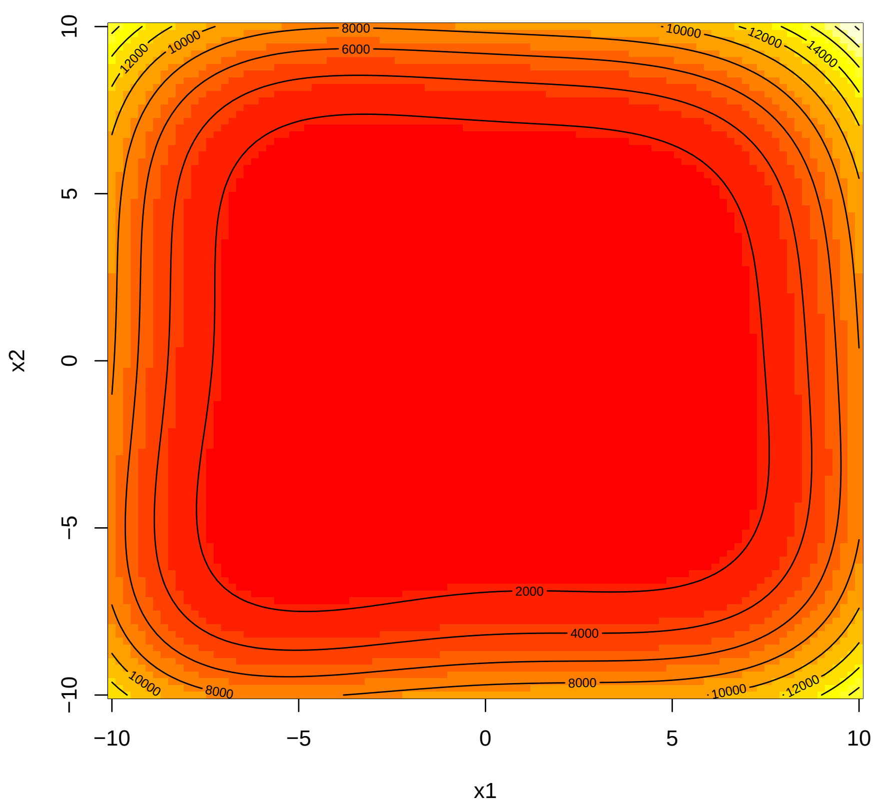
par设定初始值，fn是需要优化的目标函数，gr是梯度向量，如果是NULL则由optim()计算所得的近似值替代。lower是参数搜索的下限，默认是$-\infty$，upper是参数搜索的上限，默认是 $+\infty$。control是用来控制optim函数的一些参数，hessian=FALSE表示不需要返回海塞矩阵。在计算机里，优化的求解实质上是通过迭代算法所得。optim()提供的迭代算法主要有Nelder-Mead，BFGS，CG，L-BFGS-B，SANN。

例如：求解两元函数$g(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$ 的极值。

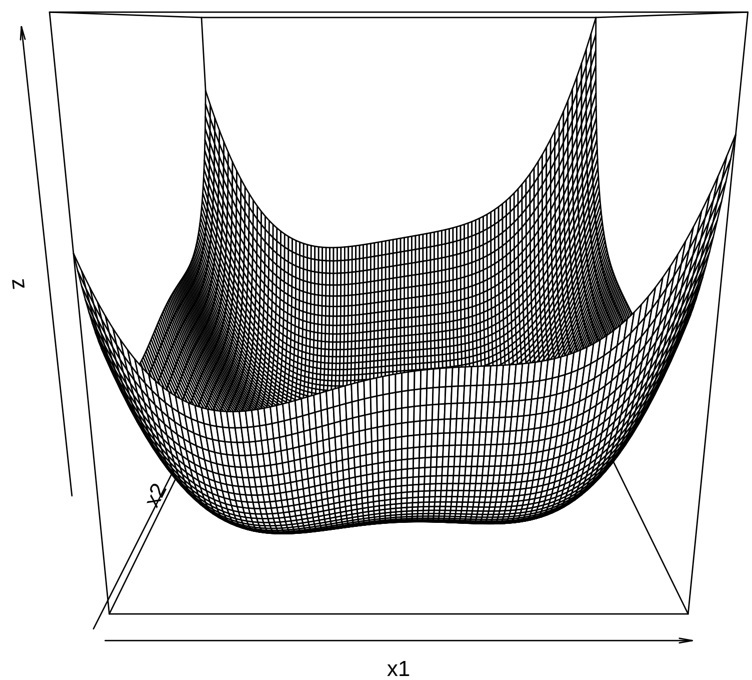
Step1 ：用R写出目标函数表达式，并画出该函数的三维图，直观上分析其极值。

```
library(arules)

## Loading required package:  Matrix
##
## Attaching package:  'arules'
##
## The following objects are masked from 'package:base':
##
##    %in%, abbreviate, write

x1=x2=seq(-10,10,length=100)
fr2=function(x1,x2){
 # x1=x[1]
#  x2=x[2]
 (x1^2+x2-11)^2+(x1+x2^2-7)^2
```

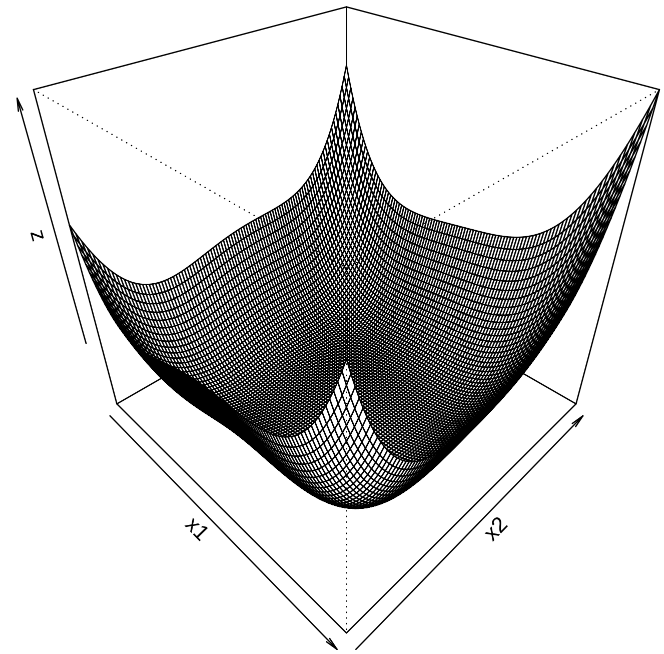
```
  }
grr=function(x ){
   x1=x[1]
   x2=x[2]                        #写一阶导表达式(梯度表达式)
c(2*(x1^2+x2-11)*2*x1+2*(x1+x2^2-7),2*(x1^2+x2-11)+2*(x1+x2^2-7)*2*x2)
}
 z=outer(x1,x2,fr2)    #求外积
  image(x1,x2,z)
  contour(x1,x2,z,add=T)      #画等高线
```



```
persp(x1,x2,z)                    #画三维图
```




```
persp(x1,x2,z,box=T,border=T,theta=45,phi=35)
```



Step2 设置优化算法迭代初始值，利用optim()进行优化求解。

```
  fr2=function(x){
  x1=x[1]
  x2=x[2]
 (x1^2+x2-11)^2+(x1+x2^2-7)^2
  }
  optim(c(-5,-5),fr2,grr) #设初始值为-5，-5，不同初始值的优化结果可能不同

## $par
## [1] -3.779382 -3.283146
##
```

```
## $value
## [1] 4.524988e-07
##
## $counts
## function gradient
##       59       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
optim(c(2,3),fr2,grr)
```

```
## $par
## [1] 3.000048 2.000034
##
## $value
## [1] 1.385542e-07
##
## $counts
## function gradient
##       57       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
optim(c(3,-2),fr2,grr)
```

```
## $par
## [1]  3.584468 -1.848056
```

```
##
## $value
## [1] 1.74435e-07
##
## $counts
## function gradient
##       57       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
optim(c(-2,3),fr2,grr)
```

```
## $par
## [1] -2.805079  3.131304
##
## $value
## [1] 5.300074e-08
##
## $counts
## function gradient
##       63       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
optim(c(-0.3,-1),fr2,grr)
```

```
## $par
```

```
## [1]  3.584498 -1.848149
##
## $value
## [1] 2.479618e-07
##
## $counts
## function gradient
##       73       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

需要注意的是不同的初始值的优化结果可能不同，可以看出，该两元函数应该有四个局部最小值。

2.0.3 约束优化

有时求解目标函数的极值是在一定的约束条件下进行。例如，求解两元函数 $g(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2, x_1 > 0, x_2 > 0$ ，的极值。此时的目标函数是在约束条件下达到最小值。

对于约束下的最优化问题，有两种方法可以求解：

一种方法是利用constrOptim（）函数直接求解，该函数的用法：

```
constrOptim(theta, f, grad, ui, ci, mu = 1e-04, control = list(),
 method = if(is.null(grad)) "Nelder-Mead" else "BFGS",
outer.iterations = 100, outer.eps = 1e-05, ...,
  hessian = FALSE)
```

theta是初始值向量，f是目标函数，grad是梯度向量，可以是空值NULL，ui是约束矩阵的左边系数矩阵，ci是约束矩阵的右边的值。比如，对于上面

的两元函数，其约束条件是$x_1 > 0, x_2 > 0$，等价于

$$\begin{cases} 1x_1 + 0x_2 & > 0 \\ 0x_1 + 1x_2 & > 0 \end{cases} \tag{1}$$

所以

$$ui = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, ci = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

例如：

```
fr2=function(x){  #编写目标函数
  x1=x[1]
   x2=x[2]
 (x1^2+x2-11)^2+(x1+x2^2-7)^2
  }
grr=function(x){
  x1=x[1]
   x2=x[2]                     ###写一阶导表达式(梯度表达式)
 c(2*(x1^2+x2-11)*2*x1+2*(x1+x2^2-7),
  2*(x1^2+x2-11)+2*(x1+x2^2-7)*2*x2)
  }
uimat=rbind(c(1,0),c(0,1))
uimat
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

```
cimat=c(0,0)
cop=constrOptim(c(0.2,0.5),fr2,grr,ui=uimat,ci=cimat)
cop
```

```
## $par
## [1] 3 2
##
## $value
```

```
## [1] 2.503875e-22
##
## $counts
## function gradient
##       47       16
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $outer.iterations
## [1] 3
##
## $barrier.value
## [1] 3.178688e-05
```

Par是目标函数达到极值时的x_1 和x_2 的取值分布为(3,2)；value是目标函数的极值，为1.066434e-22；$ counts 表示迭代过程中用到目标函数function 52次，梯度函数gradient 16次； convergence取0表示成功收敛；

另一种方法是通过参数转换为无约束下的最优化问题。将原来的约束条件$x_1 > 0, x_2 > 0$,用指数变换$x_1 = e^{z_1}, x_2 = e^{z_2}$，即这里$z_1, z_2$没有任何限制。

```
fr3=function(z){  #编写目标函数
x1=exp(z[1])
   x2=exp(z[2])
(x1^2+x2-11)^2+(x1+x2^2-7)^2
 }
grr3=function(z){

   x1=exp(z[1])
  x2=exp(z[2])                              ###写一阶导表达式(梯度表达
```

```
式)
  c(2*(x1^2+x2-11)*2*x1*exp(z[1])+2*(x1+x2^2-7)*exp(z[1]),
    2*(x1^2+x2-11)*exp(z[2])+2*(x1+x2^2-7)*2*x2*exp(z[2]))
    }
  optran=optim(c(-1.6,-0.7),fr3,grr3)   ###注意：此处最大值返回
的是z值的取值
  ##log(0.2)=-1.6,log(0.5)=-0.7
  exp(optran$par) #将z值换算回X的取值
```

发现两种方法的最有求解结果是相同的，都是在x_1, x_2分别取(3,2)时，目标函数达到最大值，说明这两种方法是等价的。此外，R还有很多函数可以做其他的优化问题求解，比如lpSolve包的lp()函数可以做线性和整数规划求解;quadprog包的solve.QP()可以做二次规划求解。

参考文献

[1] 方匡南,朱建平,姜叶飞.R数据分析——方法与案例详解（双色）.电子工业出版社,201502.

[2] 王斌会.多元统计分析及R语言建模.暨南大学出版社.201405.