函数和并行计算

温灿红

中国科学技术大学管理学院

大纲

- 函数
- 调试
- 并行计算

函数

函数

- 函数是**R**的一个对象，通过对输入参数的计算来返回一个对象作为输出。我们之前已经用过不少**R**中自带的函数，接下来学习自定义函数。
- 自定义函数可使得
    - 代码重复利用：如果一段程序需要在多处使用， 就应该将其写成一个函数，然后在多处调用。
    - 模块化设计：把编程任务分解成小的模块，每个模块用一个函数实现， 便于理解每个模块的作用， 降低了程序复杂性， 使得程序容易管理。
- 函数定义使用 `function` 定义，一般格式为

```
函数名 <- function(形式参数表) {函数体}
```

一个简单的函数

- 三次函数

```
cube <- function(x) x ^ 3
cube
```

```
## function(x) x ^ 3
```

```
mode(cube)
```

```
## [1] "function"
```

- x: 形式参数或形参（formal arguments）。 函数调用时，形参 x 得到实际值，叫做实参（actual arguments）。

```
cube(3)
```

```
## [1] 27
```

- 形参 x 得到实参3

```
cube(1:10)
```

自定义函数（一）

$$\psi(x) = \begin{cases} x^2 & \text{if } |x| \leq 1 \\ 2|x| - 1 & \text{if } |x| > 1 \end{cases}$$

```
if (x^2 < 1) {
  x^2
} else if (x >= 1) {
  2*x-1
} else {
 -2*x-1
}

ifelse(x>2 > 1, 2*abs(x) - 1, x^2)
```

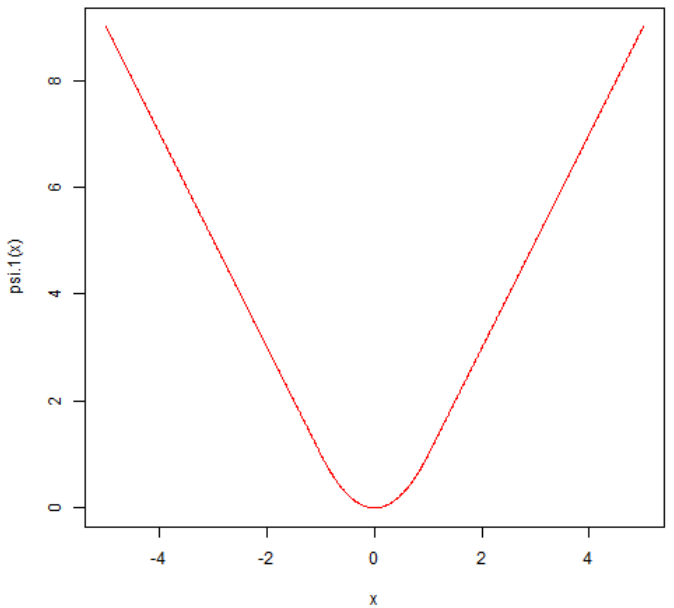
```
psi.1 <- function(x) {
  psi <- ifelse(x^2 > 1, 2*abs(x) - 1, x^2)
  return(psi)
}
```

自定义函数（二）

```
z <- c(-0.5,-5,0.9,9)
psi.1(z)
```

```
## [1]  0.25  9.00  0.81 17.00
```

```
x <- seq(from = -5, to = 5, by = 0.01)
plot(x, psi.1(x), type="l", col="red")
```



函数的结构

一个自定义R函数由三个部分组成：

- 输入，或者形式参数
- 主体部分，即函数内部要执行的代码
- 输出或返回值
    - 为了返回多个变量值， 将这些变量打包为一个列表返回即可
    - 除了用 `return(y)` 的方式在函数体的任何位置退出函数并返回 `y` 的值， 还可以用函数体的最后一个计算表达式作为返回值。

多个输入值

```
psi.2 <- function(x, c) {
  psi <- ifelse(x^2 > c^2, 2*c*abs(x)-c^2, x^2)
  return(psi)
}
identical(psi.1(z), psi.2(z,c=1))
```

```
## [1] TRUE
```

- 可通定形式参数的缺省值

```
psi.2 <- function(x, c = 1) {
  psi <- ifelse(x^2 > c^2, 2*c*abs(x)-c^2, x^2)
  return(psi)
}
identical(psi.2(z,c=1), psi.2(z))
```

```
## [1] TRUE
```

- 一般而言，在定义函数时，没有缺省值的参数写在前面， 有缺省值的参数写在后面。

函数调用

- 函数调用时最基本的调用方式是把实参与形式参数按位置对准，如

```
psi.2(z, 1)
```

```
## [1]  0.25  9.00  0.81 17.00
```

- 函数调用时全部或部分形参对应的实参可以用“形式参数名=实参”的格式给出， 这样格式给出的实参不用考虑次序。

```
identical(psi.2(x=z,c=2), psi.2(c=2,x=z))
```

```
## [1] TRUE
```

检查输入值

- 有时候不去检查输入值，而直接给出计算结果很有可能得到相悖的结论。如:

```
psi.2(x=z,c=c(1,1,1,10))
```

```
## [1]  0.25  9.00  0.81 81.00
```

```
psi.2(x=z,c=-1)
```

```
## [1]   0.25 -11.00   0.81 -19.00
```

```
psi.3 <- function(x, c = 1) {
  stopifnot(length(c) == 1,c>0) # Scale should be a single positive number
  psi <- ifelse(x^2 > c^2, 2*c*abs(x)-c^2, x^2)
  return(psi)
}
```

- stopifnot() 检查括号内的表达式是否满足，如果有不满足的则停止执行往后的命令并返回错误信息。试试看以下命令:

```
psi.3(x=z,c=c(1,1,1,10))
psi.3(x=z,c=-1)
```

输入中的懒惰求值

- 函数在调用执行时， 除非用到某个形式变量的值才求出其对应实参的值。 形参缺省值也是只有在函数运行时用到该形参的值时才求值。

```
psi.3 <- function(x, c = 1, y=ifelse(x>0, TRUE, FALSE)){
  x <- -111
  stopifnot(length(c) == 1,c>0) # Scale should be a single positive number
  print(y)
  if(y){
    ifelse(x^2 > c^2, 2*c*abs(x)-c^2, x^2)
  }else{
    ifelse(x^2 < c^2, 2*c*abs(x)-c^2, x^2)
  }
}
psi.3(5)
```

```
## [1] FALSE
```

```
## [1] 12321
```

- 虽然形参 x 输入的实参值为5, 但是这时形参 y 并没按 x = 5 被赋值为 TRUE, 而是到函数体中第二个语句才被求值， 这时 x 的值已经变成了-111， 故 y 的值是 FALSE。

输出

- 缺省返回值

```
psi.1 <- function(x) {
  psi <- ifelse(x^2 > 1, 2*abs(x) - 1, x^2)
  psi
}
```

- 返回多个对象

```
psi.4 <- function(x,c=1) {
  stopifnot(length(c) == 1,c>0)
  psi <- ifelse(x^2 > c^2, 2*c*abs(x)-c^2, x^2)
  return(list(psi = psi, x = x, c = c))
}
psi.4(z)
```

```
## $psi
## [1]  0.25  9.00  0.81 17.00
##
## $x
## [1] -0.5 -5.0  0.9  9.0
##
## $c
## [1] 1
```

副作用

- **副作用**是指在函数主体部分计算的结果，但是并没有作为返回值输出。如：
  - 打印到命令行
  - 画图
  - 保存对象到**R**文件或者外部文件。

```
psi.5 <- function(x, c = 1) {
  stopifnot(length(c) == 1,c>0)
  cat(paste0("x = ", x, ", c = ", c, "\n"))
  psi <- ifelse(x^2 > c^2, 2*c*abs(x)-c^2, x^2)
  return(list(psi = psi, x = x, c = c))
}
psi.5(1:2)
```

```
## x = 1, c = 1
##  x = 2, c = 1

## $psi
## [1] 1 3
##
## $x
## [1] 1 2
##
## $c
## [1] 1
```

形参的局部性

- 在函数被调用时， 形式参数被赋值为实际的值，如果实参是变量，形式参数可以看作实参的一个副本。
- 在函数内部对形式参数作任何修改在函数运行完成后都不影响原来的实参变量，而且函数运行完毕后形式参数不再与实际的存储空间联系。

```
x <- 7
y <- c("A","C","G","T","U")
adder <- function(y) { x<- x+y; return(x) }
adder(1)
```

```
## [1] 8
```

```
x
```

```
## [1] 7
```

```
y
```

```
## [1] "A" "C" "G" "T" "U"
```

- 在定义函数的时候，尽量避免在函数内部出现一些未在形参中定义的变量。

嵌套函数

- 允许在函数体内定义函数，但在函数内部定义的函数只能在局部使用。
- 如求解一元二次方程 $ax^2 + bx + c = 0$

```
solve.sqe <- function(x){
  fd <- function(a, b, c) b^2 - 4*a*c
  d <- fd(x[1], x[2], x[3])
  if(d >= 0){
    return( (-x[2] + c(1,-1)*sqrt(d))/(2*x[1]) )
  } else {
    return( complex(real=-x[2], imag=c(1,-1)*sqrt(-d))/(2*x[1]) )
  }
}
# 运行 fd 返回错误“Error: object 'fd' not found”
```

案例分析：随机游走

- 随机游走时间序列的产生：

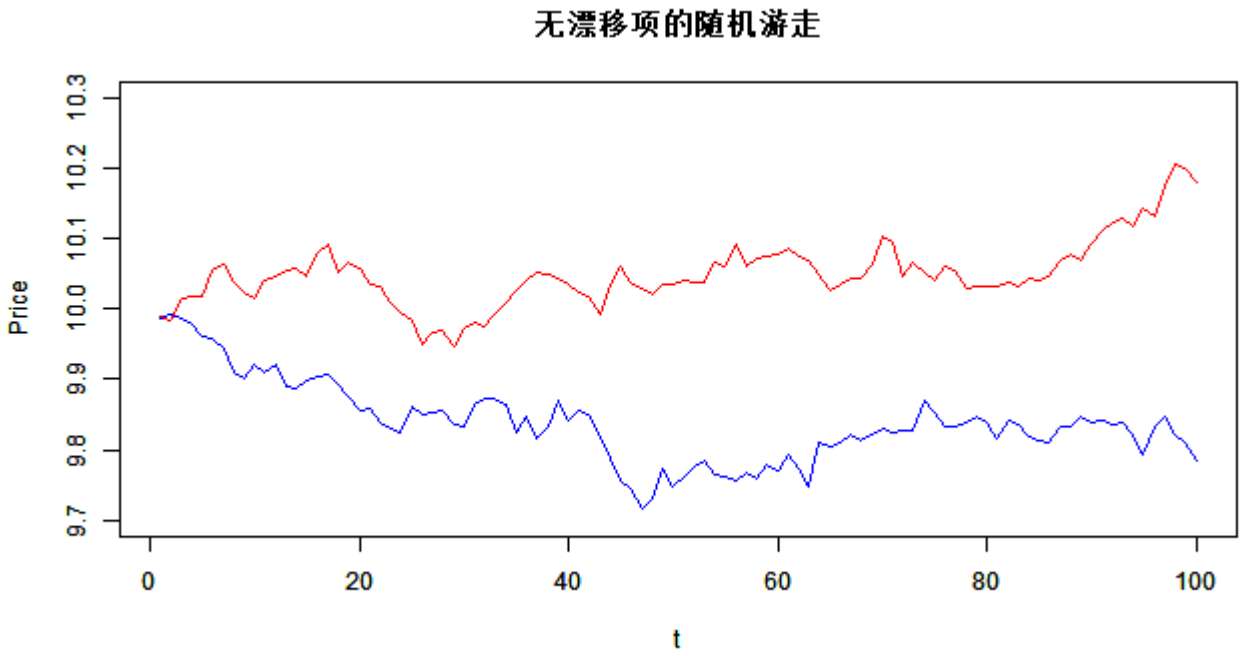
$$X_t = x_0 + t\mu + z_t,$$

其中 x_0 是初值， μ 是漂移项， z_t 是一组独立同分布的样本，均来自于标准正态分布。

```
RW <- function(N, x0, mu, variance) {
  z <- cumsum(rnorm(n = N, mean = 0, sd = sqrt(variance)))
  t <- 1:N
  x <- x0 + t*mu + z
  return(x)
}

set.seed(123)
P1<-RW(100, 10, 0, 0.0004)
P2<-RW(100, 10, 0, 0.0004)
plot(P1, main="无漂移项的随机游走", xlab="t",ylab="Price", ylim=c(9.7,10.3), typ='
par(new=T)
plot(P2, main="无漂移项的随机游走", xlab="t",ylab="Price", ylim=c(9.7,10.3),typ='l
```

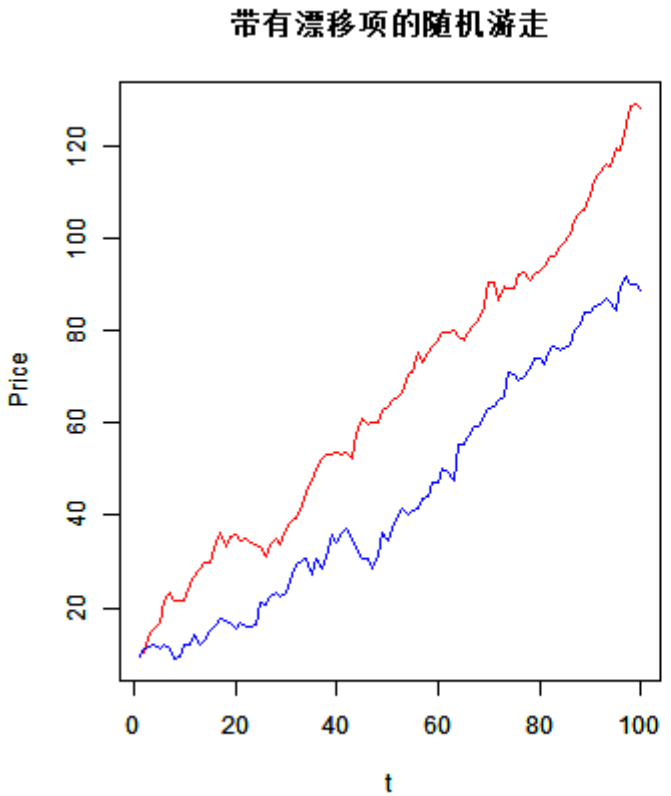
案例分析：随机游走

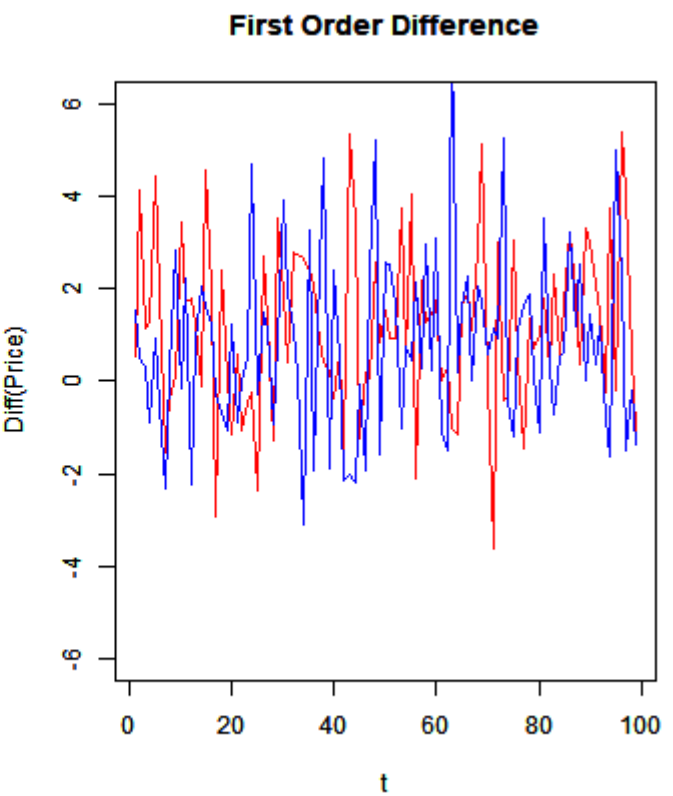


案例分析：随机游走

```
set.seed(123)
P1<-RW(100, 10, 1, 4)
P2<-RW(100, 10, 1, 4)
ran <- range(c(P1, P2))
par(mfrow=c(1,2))
plot(P1, main="带有漂移项的随机游走", xlab="t",ylab="Price", ylim = ran, type='l',
par(new=T)
plot(P2, main="带有漂移项的随机游走", xlab="t",ylab="Price",  ylim = ran, type='l'
P1_diff <- diff(P1)
plot(P1_diff, type = "l", main = "First Order Difference", col="red", xlab="t",yla
par(new=T)
P2_diff <- diff(P2)
plot(P2_diff, type = "l", main = "First Order Difference", col="blue", xlab="t",yl
```

案例分析：随机游走


带有漂移项的随机游走
Price
t
20
40
60
80
100
120
0
20
40
60
80
100


First Order Difference
Diff(Price)
t
-6
-4
-2
0
2
4
6
0
20
40
60
80
100

调试

调试的基本原则

- **确认原则**：确保代码中我们认为是真的事物实际上是真的，也就是说我们要确保输入是正确的。
- **从小处着手**：减少引起错误的程序的复杂程度， 将不必要的代码尽可能用固定的输入数据代替， 使得出错程序很短， 而且错误可重复。
- **模块化、自顶向下的调试风格**：先从出错的函数出发，一步一步追溯上去。可以用 `trackback()`找到出错的函数。
- **反漏洞**：在写函数的时候，可以设置检查自变量输入是否满足要求。如前面提到的 `stopifnot` 函数。
- 常用的调试命令：`browser()`, `debug()`, `trace()` 等
- 更多的可见《R语言编程艺术》第13章。

一个简单的函数

考虑如下函数定义:

```
f <- function(x){
  for(i in 1:n){
    s <- s + x[i]
  }
}
print(f(1:5))
```

```
## Error in f(1:5): 找不到对象'n'
```

- 简单的函数可以直接检查发现错误， 用cat, print 等输出中间结果查找错误。

调试工具 debug()

- 在调用函数前输入debug(foo)命令，可在下面实际调用时进入调试模式。

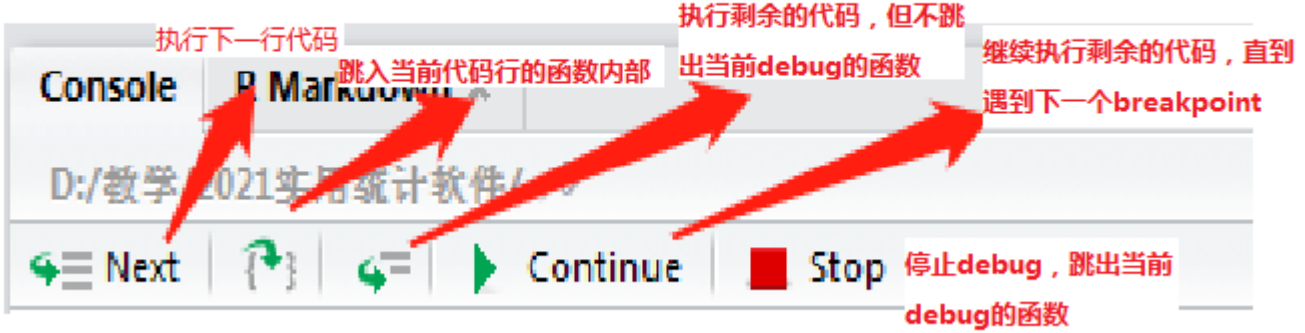
```
debug(f)
f(1:5)
```

```
## debugging in: f(1:5)
## debug在<text>#1: {
##     for (i in 1:n) {
##         s <- s + x[i]
##     }
## }
## debug在<text>#2: for (i in 1:n) {
##     s <- s + x[i]
## }

## Error in f(1:5): 找不到对象'n'
```

调试工具 debug()

```
> debug(f)
> f(1:5)
debugging in: f(1:5)
debug at #1: {
    for (i in 1:n) {
        s <- s + x[i]
    }
}
Browse[2]> ls()
[1] "x"
Browse[2]> print(n)
Error in print(n) : object 'n' not found
Browse[2]> print(s)
[1] "abc" "A"   "1"   NA    "男"
Browse[2]> n
debug at #2: for (i in 1:n) {
    s <- s + x[i]
}
```



调试工具 browser()

- 在程序中插入 browser() 函数的调用， 可以进入跟踪调试状态， 可以实时地查看甚至修改运行时变量的值。

```
f <- function(x){
  browser()
  for(i in 1:n){
    s <- s + x[i]
  }
}
print(f(1:5))
```

```
## Called from: f(1:5)
## debug在<text>#3: for (i in 1:n) {
##     s <- s + x[i]
## }

## Error in f(1:5): 找不到对象'n'
```

条件断点

- 用 browser() 函数与 if 结构配合可以制作条件断点。
- 如在调试带有循环的程序时，发现错误发生在循环内，如果从循环开始就跟踪运行，会浪费许多时间。 设已知错误发生在循环变量 i 等于501的时候， 就可以在循环内插入：

```
if(i == 501) browser()
```

一个简单的函数

1. n 未定义
2. s 未初值化
3. 没有返回值

- 修正如下：

```
f <- function(x){
  s <- 0
  for(i in seq_along(x)){
    s <- s + x[i]
  }
  s
}
```

并行计算

并行计算

- 现代桌面电脑和笔记本电脑的CPU通常有多个核心或虚拟核心（线程），如2核心或4虚拟核心。通常**R**运行并不能利用全部的CPU能力，仅能利用其中的一个虚拟核心。
- 利用多台计算机、多个CPU、CPU中的多核心和多线程同时完成一个计算任务称为并行计算。
- parallel、snowfall

例子：完全不相互依赖的并行计算

- 考虑如下的计算问题：

$$S_{k,n} = \sum_{i=1}^{n} \frac{1}{i^k}.$$

- 令 $n = 5000000, k = 2, \ldots, 21$，分别计算 $S_{k,n}$值并输出

```
f10 <- function(k, n){
  s <- 0.0
  for(i in seq(n)) s <- s + 1/i^k
  s
}
n<- 5000000
nk <- 20

cur.time <- Sys.time()
v <- sapply(2:(nk+1), function(k) f10(k, n))
Sys.time() - cur.time
```

```
## Time difference of 7.426559 secs
```

并行版本

```
library(snowfall)
```

```
## 载入需要的程辑包：snow
```

```
cur.time <- Sys.time()
sfInit( parallel=TRUE, cpus=4)
```

```
## R Version:  R version 4.2.3 (2023-03-15 ucrt)

## snowfall 1.84-6.3 initialized (using snow 0.4-4): parallel execution on 4 CPUs.
```

```
sfExport("f10", "n")
v.sf <- sfSapply(2:(nk+1), function(k) f10(k, n))
sfStop()
```

```
##
## Stopping cluster
```

```
Sys.time() - cur.time
```

```
## Time difference of 2.978629 secs
```

代码解读（一）

- 先查看本计算机的虚拟核心（线程）数：

```
sfCpus()
```

```
## Calling a snowfall function without calling 'sfInit' first or after sfStop().
## 'sfInit()' is called now.

## snowfall 1.84-6.3 initialized: sequential execution, one CPU.

## [1] 4
```

- 发现有4个节点，用sfInit()建立临时的有4个节点的单机集群：

```
sfInit( parallel=TRUE, cpus=4)
```

代码解读（二）

- 用sfExport()把计算所依赖的对象预先传送到每个节点

```
sfExport("f10", "n")
```

- 类似的函数有
  - sfLibrary：把计算所依赖的包导入到每个节点
  - sfSource：把计算所依赖的源代码文件导入到每个节点
  - sfExportAll：导入所有的全局变量

代码解读（三）

- sfSapply()函数是sapply函数的并行版本，用她来对 k 并行地循环：

```
v.sf <- sfSapply(2:(nk+1), function(k) f10(k, n))
```

- 类似的函数有
  - sfLapply：lapply函数的并行版本
  - sfAapply：apply函数的并行版本
-
- 并行执行结束后， 需要解散临时的集群， 否则可能会有内存泄漏：

```
sfStop()
```

案例分析：核密度估计

- 假设连续随机变量 X 的概率密度函数为 f，则我们有

$$f(x) = \lim_{h \to 0} \frac{1}{2h} P(x - h < X < x + h)$$

- 那么假设有样本 $x_1, x_n, \ldots, x_n$ 为一组独立同分布的样本，一个关于密度函数 f 的朴素估计为

$$\widehat{f}(x) = \frac{1}{2nh} \#\{ x_i \in (x-h, x+h) \}$$

而这个其实就是以 h 为窗宽的直方图。也可以等价写成

$$\hat{f}(x) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{2h} I(x_i \in (x - h, x + h))$$

- 更一般地，我们可以替换不连续的示性函数为连续的函数。一般的核密度估计定义为：

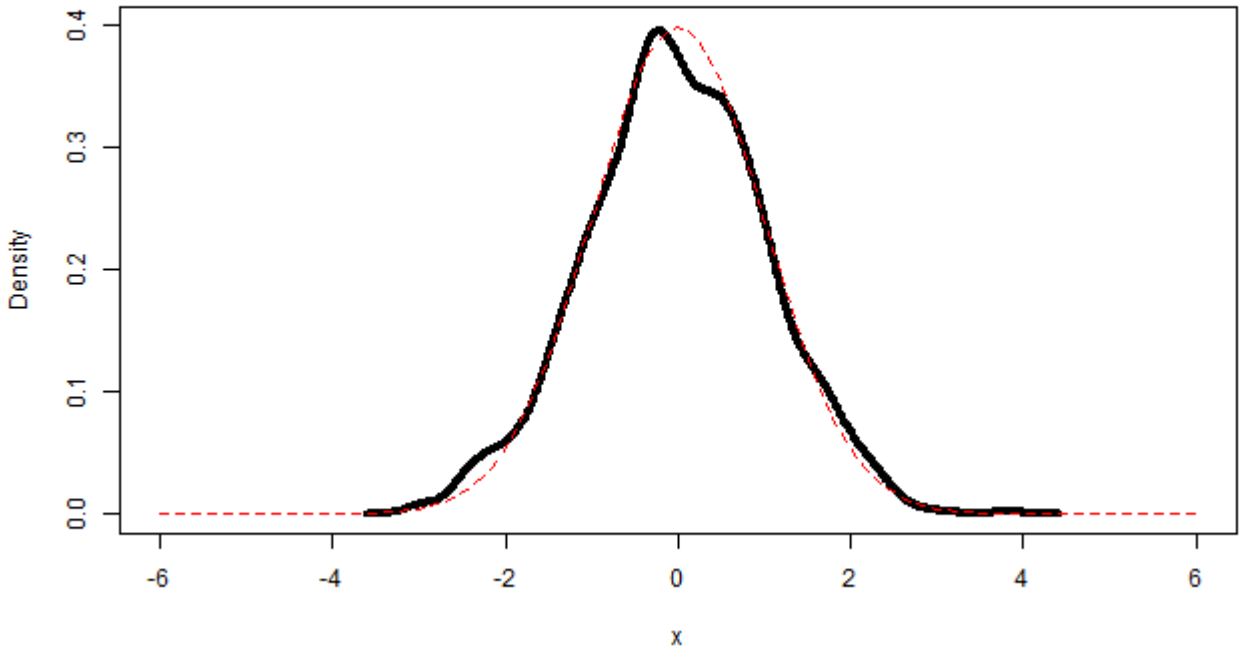
$$\hat{f}(x) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{h} K\left(\frac{x - x_i}{h} \right).$$

案例分析：核密度估计

```r
# x: 要进行核密度估计的数据，数值型向量
# h: 窗宽
# m: 核密度估计的取样网格点
kern_dens <- function(x, h, m = 512) {
  rg <- range(x)
  xx <- seq(rg[1] - 3 * h, rg[2] + 3 * h, length.out = m)
  y <- numeric(m)
  for (i in seq_along(xx))
    for (j in seq_along(x))
      y[i] <- y[i] + exp(- (xx[i] - x[j])^2 / (2 * h^2))
  y <- y / (sqrt(2 * pi) * h * length(x))
  list(x = xx, y = y)
}
```

案例分析：核密度估计

```
set.seed(1)
psi <- rnorm(1000)
f_hat <- kern_dens(psi, 0.2)
plot(f_hat, type = "l", lwd = 4, xlab = "x", ylab = "Density", xlim = c(-6,6), yli
curve(dnorm, from = -6, to = 6, add = TRUE, col = "red", lty = 2)
```



案例分析：核密度估计

- 向量化：简单向量化

```
kern_dens_vec <- function(x, h, m = 512) {
  rg <- range(x)
  xx <- seq(rg[1] - 3 * h, rg[2] + 3 * h, length.out = m)
  y <- numeric(m)
  const <- (sqrt(2 * pi) * h * length(x))
  for (i in seq_along(xx))
      y[i] <- sum(exp(-(xx[i] - x)^2 / (2 * h^2))) / const
  list(x = xx, y = y)
}
```

案例分析：核密度估计

- 向量化：apply

```
kern_dens_apply <- function(x, h, m = 512) {
  rg <- range(x)
  xx <- seq(rg[1] - 3 * h, rg[2] + 3 * h, length.out = m)
  const <- sqrt(2 * pi) * h * length(x)
  y <- sapply(xx, function(z) sum(exp(-(z - x)^2 / (2 * h^2))) / const)
  list(x = xx, y = y)
}
```

案例分析：核密度估计

- 并行计算

```
library(snowfall)

kern_dens_snow <- function(x, h, m = 512) {
  rg <- range(x)
  xx <- seq(rg[1] - 3 * h, rg[2] + 3 * h, length.out = m)
  const <- sqrt(2 * pi) * h * length(x)

  sfExport("x", "const", "h")
  y <- sfSapply(xx, function(z) sum(exp(-(z - x)^2 / (2 * h^2))) / const)

  list(x = xx, y = y)
}
```

案例分析：核密度估计

```
system.time(kern_dens(psi, 0.2))
```

```
## 用户 系统 流逝
## 0.10 0.00 0.09
```

```
system.time(kern_dens_vec(psi, 0.2))
```

```
## 用户 系统 流逝
## 0.04 0.00 0.05
```

```
system.time(kern_dens_apply(psi, 0.2))
```

```
## 用户 系统 流逝
## 0.05 0.00 0.05
```

案例分析：核密度估计

```
sfInit( parallel=TRUE, cpus=4)
```

```
## Explicit sfStop() is missing: stop now.
```

```
## snowfall 1.84-6.3 initialized (using snow 0.4-4): parallel execution on 4 CPUs.
```

```
system.time(kern_dens_snow(psi, 0.2))
```

```
##  用户  系统  流逝
##  0.02  0.00  0.01
```

```
sfStop()
```

```
##
## Stopping cluster
```

案例分析：核密度估计

```
library(microbenchmark)
sfInit( parallel=TRUE, cpus=4)
```

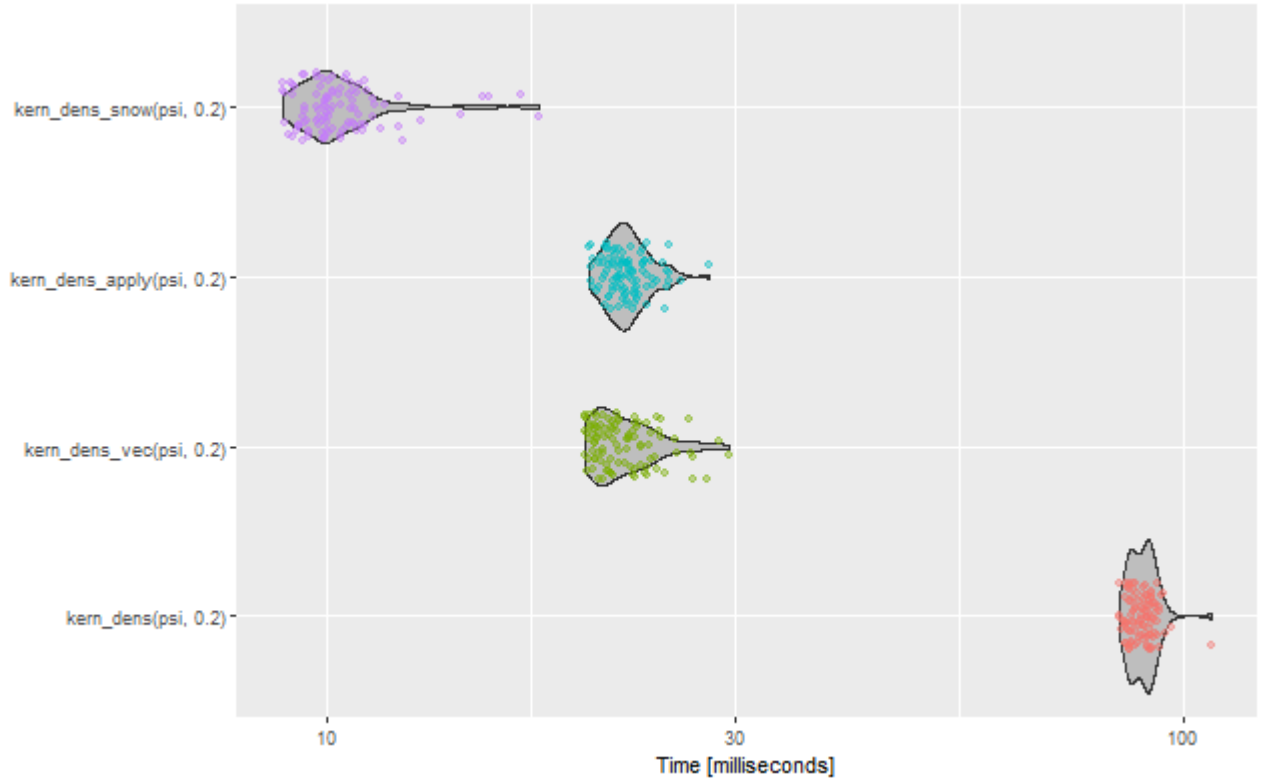
```
## snowfall 1.84-6.3 initialized (using snow 0.4-4): parallel execution on 4 CPUs.
```

```
kern_bench <- microbenchmark(
  kern_dens(psi, 0.2),
  kern_dens_vec(psi, 0.2),
  kern_dens_apply(psi, 0.2),
  kern_dens_snow(psi, 0.2)
)
sfStop()
```

```
##
## Stopping cluster
```

案例分析：核密度估计

```
library(ggplot2)
autoplot(kern_bench) +
  geom_jitter(position = position_jitter(0.2, 0), aes(color = expr), alpha = 0.4) +
  aes(fill = I("gray")) + theme(legend.position = "none")
```



谢 谢