Section functions to a smaller domain with `section_fun()` in the `doBy` package

Søren Højsgaard

doBy version 4.6.21 as of 2024-04-29

Contents

1 Introduction

The **doBy** package contains a variety of utility functions. This working document describes some of these functions. The package originally grew out of a need to calculate groupwise summary statistics (much in the spirit of `PROC SUMMARY` of the SAS system), but today the package contains many different utilities.

2 Section a functions domain: `section_fun()`

Let $f(x, y) = x + y$. Then $f_x(y) = f(10, y)$ is a section of f to be a function of y alone.

More generally, let E be a subset of the cartesian product $X \times Y$ where X and Y are some sets. Consider a function $f(x, y)$ defined on E. Then for any $x \in X$, the section of E defined by x (denoted E_x) is the set of y's in Y such that (x, y) is in E, i.e.

$$E_x = \{y \in Y | (x, y) \in E\}$$

Correspondingly, the section of $f(x, y)$ defined by x is the function f_x defined on E_x given by $f_x(y) = f(x, y)$.

There are the following approaches:

1) insert the section values as default values in the function definition (default), 2) insert the section values in the function body, 3) store the section values in an auxillary environment.

Consider this function:

```
> fun  <- function(a, b, c=4, d=9){
      a + b + c + d
  }
```

```
> fun_def <- section_fun(fun, list(b=7, d=10))
> fun_def

## function (a, c = 4, b = 7, d = 10)
## {
##     a + b + c + d
## }

> fun_body <- section_fun(fun, list(b=7, d=10), method="sub")
> fun_body

## function(a, c=4)
## {
##  ## section
##  b = 7;
##  d = 10
##  ## section (end)
##  a + b + c + d
## }

> fun_env <- section_fun(fun, list(b=7, d=10), method = "env")
> fun_env

## function (a, c = 4)
## {
##     . <- "use get_section(function_name) to see section"
##     . <- "use get_fun(function_name) to see original function"
##     args <- arg_getter()
##     do.call(fun, args)
## }
## <environment: 0x56bb6c5a77a0>
```

In the last case, we can see the section and the original function definition as:

```
> get_section(fun_env)

## $b
## [1] 7
##
## $d
## [1] 10
```

```
> ## same as: attr(fun_env, "arg_env")$args
> get_fun(fun_env)

## function(a, b, c=4, d=9){
##     a + b + c + d
## }

> ## same as: environment(fun_env)$fun
```

We get:

```
> fun(a=10, b=7, c=5, d=10)

## [1] 32

> fun_def(a=10, c=5)

## [1] 32

> fun_body(a=10, c=5)

## [1] 32

> fun_env(a=10, c=5)

## [1] 32
```

3 Example: Benchmarking

Consider a simple task: Creating and inverting Toeplitz matrices for increasing dimensions:

```
> n <- 4
> toeplitz(1:n)

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

A naive implementation is

```
> inv_toep <- function(n) {
      solve(toeplitz(1:n))
  }
> inv_toep(4)

##      [,1] [,2] [,3] [,4]
## [1,] -0.4  0.5  0.0  0.1
## [2,]  0.5 -1.0  0.5  0.0
## [3,]  0.0  0.5 -1.0  0.5
## [4,]  0.1  0.0  0.5 -0.4
```

We can benchmark timing for different values of n as

```
> library(microbenchmark)
> microbenchmark(
      inv_toep(4), inv_toep(8), inv_toep(16),
      inv_toep(32), inv_toep(64),
      times=5
  )

## Unit: microseconds
##          expr    min     lq   mean median     uq     max neval cld
##   inv_toep(4)  12.30  13.00  15.21  13.08  14.84   22.84     5   a
##   inv_toep(8)  13.99  14.33  15.02  14.99  15.71   16.10     5   a
##  inv_toep(16)  21.22  21.27  22.02  21.91  21.96   23.72     5   a
##  inv_toep(32)  48.62  48.90  50.40  49.16  50.76   54.59     5   a
##  inv_toep(64) 160.34 169.51 475.13 175.16 179.32 1691.34     5   a
```

However, it is tedious (and hence error prone) to write these function calls.

A programmatic approach using `section_fun` is as follows: First create a list of sectioned functions:

```
> n.vec  <- c(3, 4, 5)
> fun_list <- lapply(n.vec,
                     function(ni){
                         section_fun(inv_toep, list(n=ni))}
                     )
```

We can inspect and evaluate each / all functions as:

```
> fun_list[[1]]

## function (n = 3)
## {
```

```
##     solve(toeplitz(1:n))
## }

> fun_list[[1]]()

##        [,1] [,2]   [,3]
## [1,] -0.375  0.5  0.125
## [2,]  0.500 -1.0  0.500
## [3,]  0.125  0.5 -0.375
```

To use the list of functions in connection with microbenchmark we bquote all functions using

```
> bquote_list <- function(fnlist){
      lapply(fnlist, function(g) {
          bquote(.(g)())
      }
      )
  }
```

We get:

```
> bq_fun_list <- bquote_list(fun_list)
> bq_fun_list[[1]]

## (function (n = 3)
## {
##     solve(toeplitz(1:n))
## })()

> ## Evaluate one:
> eval(bq_fun_list[[1]])

##        [,1] [,2]   [,3]
## [1,] -0.375  0.5  0.125
## [2,]  0.500 -1.0  0.500
## [3,]  0.125  0.5 -0.375

> ## Evaluate all:
> ## lapply(bq_fun_list, eval)
```

To use microbenchmark we must name the elements of the list:

```
> names(bq_fun_list) <- n.vec
> microbenchmark(
    list  = bq_fun_list,
    times = 5
  )

## Unit: microseconds
##  expr   min    lq  mean median    uq   max neval cld
##     3 11.07 11.12 25.14  11.37 13.47 78.69     5   a
##     4 11.56 11.83 14.29  12.58 14.30 21.18     5   a
##     5 12.19 12.31 15.38  13.24 15.32 23.84     5   a
```

Running the code below provides a benchmark of the different ways of sectioning in terms of speed.

```
> n.vec  <- seq(20, 80, by=20)
> fun_def <- lapply(n.vec,
                    function(ni){
                        section_fun(inv_toep, list(n=ni), method="def")}
                    )
> fun_body <- lapply(n.vec,
                    function(ni){
                        section_fun(inv_toep, list(n=ni), method="sub")}
                    )
> fun_env <- lapply(n.vec,
                    function(ni){
                        section_fun(inv_toep, list(n=ni), method="env")}
                    )
>
> bq_fun_list <- bquote_list(c(fun_def, fun_body, fun_env))
> names(bq_fun_list) <- paste0(rep(c("def", "body", "env"), each=length(n.vec)), rep(n.vec, 3
>
> mb <- microbenchmark(
    list  = bq_fun_list,
    times = 2
  )
```