Rcpp use case: segmentation by dynamic programming

Pierre Neuvial

December 11, 2015

Prerequisite: plain `R` implementation of segmentation by dynamic programming (see last practicals).

```
source("dpseg.R")
dpseg
```

```
function (y, K)
{
    n <- length(y)
    J <- getJ(y)
    dp <- getVandBkp(J, K)
    V <- dp$V
    bkp <- dp$bkp
    res.bkp <- backtrack(bkp)
    res.rse <- V[, n]
    list(bkpList = res.bkp, rse = res.rse, V = V)
}
```

1 Which part of the code takes time?

You can use `Rprof`, or the `profr` package. For example:

```
y <- rnorm(2e3)
Rprof(file="dpseg.Rout")
res <- dpseg(y, K=10)
Rprof(NULL)
sr <- summaryRprof("dpseg.Rout")
head(sr$by.self, 10)
```

```
##              self.time self.pct total.time total.pct
## "getJ"            6.52    74.43       7.70     87.90
## "getVandBkp"      0.70     7.99       1.06     12.10
## "+"               0.56     6.39       0.56      6.39
## "-"               0.28     3.20       0.28      3.20
## "^"               0.20     2.28       0.20      2.28
## ":"               0.18     2.05       0.18      2.05
## "("               0.14     1.60       0.14      1.60
## "matrix"          0.08     0.91       0.08      0.91
## "min"             0.04     0.46       0.04      0.46
## "which.min"       0.04     0.46       0.04      0.46
```

Most of the time is spent in `getJ`. This is even worse for larger signals (not shown here because this file would take too long to compile).

```r
library("rbenchmark")
benchmark(getJ(rnorm(1e2)), getJ(rnorm(1e3)), getJ(rnorm(2e3)), replications=1)[, 1:4]
```

```
##                test replications elapsed relative
## 1  getJ(rnorm(100))            1   0.019    1.000
## 2 getJ(rnorm(1000))            1   2.124  111.789
## 3 getJ(rnorm(2000))            1   7.750  407.895
```

2 Implementing a C++ version of `getJ`

We use `Rcpp` to re-implement the following part of 'getJ':

```r
J <- matrix(NA, ncol=n, nrow=n)
for (ii in 1:n) {
    for(jj in ii:n) {
        J[ii, jj] <- T[jj+1]-T[ii] - (S[jj+1]-S[ii])^2/(jj-ii+1)
    } ## for (jj ...
} ## for (ii ...
```

More precisely, we write a `cpp` file that implements a function `fillJcpp` taking as input the vectors `T` and `S`, and returning the corresponding matrix `J`. See file `getJ.cpp`.

```r
Rcpp::sourceCpp('getJ.cpp')
```

```
##
## > y <- rnorm(5)
##
## > S <- c(0, cumsum(y))
##
## > T <- c(0, cumsum(y^2))
##
## > fillJcpp(S, T)
##      [,1]         [,2]         [,3]          [,4]         [,5]
## [1,]    0 6.220261e-01 1.745765e+00  2.085716e+00 3.097752e+00
## [2,]    0 1.942890e-16 2.742621e-01  2.953907e-01 7.044114e-01
## [3,]    0 0.000000e+00 1.387779e-17  1.848720e-02 2.012995e-01
## [4,]    0 0.000000e+00 0.000000e+00 -3.122502e-16 1.920775e-01
## [5,]    0 0.000000e+00 0.000000e+00  0.000000e+00 2.220446e-16
##
## > getJ(y)
##      [,1]         [,2]         [,3]          [,4]         [,5]
## [1,]    0 6.220261e-01 1.745765e+00  2.085716e+00 3.097752e+00
## [2,]   NA 1.942890e-16 2.742621e-01  2.953907e-01 7.044114e-01
## [3,]   NA           NA 1.387779e-17  1.848720e-02 2.012995e-01
## [4,]   NA           NA           NA -3.122502e-16 1.920775e-01
## [5,]   NA           NA           NA            NA 2.220446e-16
##
## > fillJcpp(S, T) - getJ(y)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
```

```
## [2,]   NA    0    0    0    0
## [3,]   NA   NA    0    0    0
## [4,]   NA   NA   NA    0    0
## [5,]   NA   NA   NA   NA    0
```

This function can readily be used within our code:

```
getJcpp <- function(y) {
    n <- length(y)
    S <- c(0, cumsum(y))
    T <- c(0, cumsum(y^2))
    J <- fillJcpp(S, T)
    return(J)
}
```

2.1 Rcpp vs R: same results ?

```
y <- rnorm(5)
Jc <- getJcpp(y)
J <- getJ(y)
max(abs(J-Jc), na.rm=TRUE)
```

```
## [1] 0
```

2.2 Some benchmarking

```
n <- 2e3
benchmark(getJ(rnorm(n)), getJcpp(rnorm(n)), getJcpp(rnorm(10*n)),replications=1)[, 1:4]
```

```
##                      test replications elapsed relative
## 1          getJ(rnorm(n))            1   6.797   95.732
## 3 getJcpp(rnorm(10 * n))            1  19.377  272.915
## 2       getJcpp(rnorm(n))            1   0.071    1.000
```

The `C++` version is approximately **100 times faster** than the original one. This is because we were using two nested loops in `R`, which is typically quite slow.

3 Comparing the two implementations

3.1 Integrating the `C++` initialization into DP segmentation

We are ready to implement of version of dynamic programming where `C++` is used for initialization:

```
dpsegCpp <- function(y, K){
    n <- length(y)

    ## Compute the k*k matrix J such that J[i,j] for i<=j is the RSE
    ## when intervals i to j are merged
    J <- getJcpp(y)  ## Initialization

    ## Dynamic programming
    dp <- getVandBkp(J, K)
    V <- dp$V
    ## V[i,j] is the best RSE for segmenting intervals 1 to j
    ## with at most i-1 change points
    bkp <- dp$bkp
    ## bkp[i, j] is the *last* bkp of the best segmentation of [1:j] in i segments

    ## Optimal segmentation
    res.bkp <- backtrack(bkp)

    ## RSE as a function of number of change-points
    res.rse <- V[, n]
    ## Optimal number of change points

    list(bkpList=res.bkp, ##<< A list of vectors of change point positions for the best model with k cha
         rse=res.rse, ##<< A vector of K+1 residual squared errors
         V=V) ##<< V[i,j] is the best RSE for segmenting intervals 1 to j
}
```

3.2 Rcpp vs R: same results ?

```
y <- rnorm(1e3)
resR <- dpseg(y, K=10)
resCpp <- dpsegCpp(y, K=10)
identical(resR, resCpp)
```

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

3.3 Some more benchmarking

```
y <- rnorm(1e3)

Rprof(file="dpseg.Rout")
res <- dpseg(y, K=10)
Rprof(NULL)
sr <- summaryRprof("dpseg.Rout")
head(sr$by.self, 10)
```

```
##                self.time self.pct total.time total.pct
## "getJ"              1.44       72       1.74        87
```

```
## "getVandBkp"      0.16       8      0.26       13
## "+"               0.12       6      0.12        6
## "-"               0.10       5      0.10        5
## "^"               0.10       5      0.10        5
## ":"               0.04       2      0.04        2
## "which.min"       0.04       2      0.04        2
```

```
head(sr$by.total, 10)
```

```
##                         total.time total.pct self.time self.pct
## "<Anonymous>"                    2       100         0        0
## "block_exec"                     2       100         0        0
## "call_block"                     2       100         0        0
## "dpseg"                          2       100         0        0
## "eval"                           2       100         0        0
## "evaluate_call"                  2       100         0        0
## "handle"                         2       100         0        0
## "in_dir"                         2       100         0        0
## "process_file"                   2       100         0        0
## "process_group.block"            2       100         0        0
```

```
Rprof(file="dpseg,cpp.Rout")
res <- dpsegCpp(y, K=10)
Rprof(NULL)
sr <- summaryRprof("dpseg,cpp.Rout")
head(sr$by.self, 10)
```

```
##                self.time self.pct total.time total.pct
## "getVandBkp"        0.22    64.71       0.34    100.00
## "+"                 0.06    17.65       0.06     17.65
## ":"                 0.04    11.76       0.04     11.76
## "min"               0.02     5.88       0.02      5.88
```

Now with a larger signal

```
y <- rnorm(1e4)
Rprof(file="dpseg4,cpp.Rout")
res <- dpsegCpp(y, K=10)
Rprof(NULL)
sr <- summaryRprof("dpseg4,cpp.Rout")
head(sr$by.self, 10)
```

```
##                 self.time self.pct total.time total.pct
## "getVandBkp"        15.44    60.84      23.44     92.36
## ":"                  4.32    17.02       4.32     17.02
## "+"                  3.16    12.45       3.16     12.45
## "<Anonymous>"        1.94     7.64      25.38    100.00
## "which.min"          0.28     1.10       0.28      1.10
## "min"                0.24     0.95       0.24      0.95
```

The next step that can be improved is the calculation of the `V` and `bkp` matrices in function `getVandBkp`.