Rcpp use case: segmentation by dynamic programming

Pierre Neuvial

December 11, 2015

1 Motivation

Last week we have written a plain `R` implementation of segmentation by dynamic programming:

```
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)
}
```

This function makes use of three intermediate functions:

- `getJ` computes the $n \times n$ matrix `J` such that `J[i,j]` for `i` $\leq$ `j` is the Residual Squared Error (RSE) of the segmentation with only *one* segment (no breakpoint) between `i` and `j`;
- `getVandBkp` computes from `J` the matrices `V` ($K + 1 \times n$) and `bkp` ($K \times n$), where `V[i,j]` is the best RSE for segmenting intervals `1` to `j` with at most `i-1` change points, and `bkp[i, j]` is the *last* bkp of the best segmentation of `[1:j]` in `i` segments;
- `backtrack` retrieves the optimal segmentation of `[1,n]` in `k` segments for all `k` from `bkp`.

The `dpseg` runs quickly for signals of length 1000. For a signal of length 10^4, the segmentation in 10 segments takes approximately 4 minutes on a standard laptop. As the time complexity is quadratic, we expect a signal of length 10^5 to be segmented in approximately 400 minutes.

The goal of this section is to identify which part of the code takes the longest, and to use `Rcpp` (that is, `C++` code interfaced with `R`) in order to speed up this part.

- Diagnostic: which part of the code takes time?
- Use Rcpp to obtain a faster implementation of this part of the code
- What is the relative gain in computing time?
- Incorporate this implementation into the whole dynamic programming function