7 Loose Ends

7.1 More Commands for Subsetting

Suppose you have a vector of observations for which you want to transform all entries satisfying a specified condition according to a rule you specify. For example, consider the following data on final exam grades from an undergraduate class:

```
Final410 <- c(75,93,71,71,50,71,57,53,74,71,100,92,74,93,95,
                68,70,55,100,29,78,63,34,55)
```

If we want to change grades by adding 3 points to all scores below 65, then here are four distinct ways which do **not** use for-loops:

```
> x <- Final410; x[x<65] <- x[x<65]+3;
> x <- Final410 + 3*as.numeric(Final410<65)
> x <- replace(Final410, Final410<65, Final410[Final410<65]+3)
> x <- ifelse(Final410<65, Final410+3, Final410)
```

All four command-lines give the same result. I find the last one the most attractive conceptually: *ifelse* is a very nice command for adjusting pieces of vectors. Note that the third argument in the *replace* function above must designate the components needed to do the replacement and must have exactly the same length as the number of entries satisfying the condition specified by the second argument ! (The command

```
> x <- replace(Final410, Final410<65, Final410+3)
```

definitely gives a different result.)

7.2 Special Syntax for Parallelizing

Once you get used to avoiding for-loops by parallelizing, you will naturally try to write all of your expressions so that they make sense and give componentwise correct results when applied to vectors. In a few cases, this requires special syntax. For example, *min(x,y)* denotes the smaller of the two numbers x, y, but if x and y are vectors of the same length, then *min(x,y)* gives the same result as *min(c(x,y))*, which is the smallest single entry in the combined vector. If instead you want the vector of coordinatewise smaller entries *x[i], y[i]*, then the command is *pmin(x,y)*. (Similarly for *max* use instead *pmax*. Another example is: ($x < y$ && $x >= 3$), which has the natural Boolean interpretation if x and y are scalar, but you must use & in place of && if you want the componentwise correct Boolean vector.

7.3 Meaning of (Smoothing) Splines

The splines which we use in speeding up interpolation and inversion of functions are provided by Splus functions *smooth.spline* and *predict.smooth.spline*. The mathematical definition of these functions is as the solution of the following optimization problem. Suppose that data-pairs $\{(x_i, y_i)\}_{i=1}^n$ are given, with $x_i \in [a, b]$ and a, b known, and that a subset $\mathcal{K} \subset \{x_i\}_{i=1}^n$ and a positive constant λ are specified. The problem is to find the *continuously differentiable* function $s : [a, b] \mapsto \mathbf{R}$ satisfying $s(x_i) = y_i \; \forall x_i \in \mathcal{K}$ to

$$\text{minimize} \qquad \sum_{i=1}^n (y_i - s(x_i))^2 + \lambda \int_a^b (s''(x))^2 \, dx$$

If there were no knots at all ($\mathcal{K} = \emptyset$), then the solution is obviously the least-squares line. More generally, it can be shown that the solution $s(\cdot)$ is a piecewise cubic polynomial, which is also called a *cubic spline*. For given λ the solution exhibits more smoothing and less accuracy in satisfying $s(x_i) = y_i$ when the set of knots becauses smaller, and for a fixed set of knots the solution exhibits more smoothing and less accuracy in satisfying $s(x_i) = y_i$ for $x_i \notin \mathcal{K}$ when λ is made larger. In general, it works quite well to use a very small *spar* parameter (which is proportional to λ in a way which is described clearly in the online documentation to *smooth.spline*) when the (x_i, y_i) pairs are believed to pass close to a very smooth curve. But you may have to try a couple of cases and plot the resulting *predict.smooth.spline* function to see whether you have achieved the desired degree of visual smoothness.

Here is a little demonstration that *smooth.spline* and *predict.smooth.spline* produce a piecewise cubic polynomial:

```
> x <- runif(10)
> y <- 4*x^5 - 3*x^2 +2
> tmpspl <- smooth.spline(x,y,spar=1.e-6,all.knots=T)
> sort(x)[8:9]
[1] 0.5146857 0.6923524
## Since there are no points between  .52  and .65, let's look at
## the spline-function on that interval !!
>  z <- predict.smooth.spline(tmpspl, .52+(0:10)*.01)$y
> var(diff(diff(diff(z))))  ### = 1e-30
```

For a vector $\mathbf{z}$, *diff(*$\mathbf{z}$*)* is a vector of first-differences with entries $z_i - z_{i-1},\ i \geq 2$. The cubic nature of the function at the points $.52, .53, \ldots, .62$ is shown by the fact that the third differences are constant.