Statistical Computing with **R**

Eric Slud, Math. Dept., UMCP

October 21, 2009

Overview of Course

This course was originally developed jointly with Benjamin Kedem and Paul Smith. It consists of modules as indicated on the Course Syllabus. These fall roughly into three main headings:

(A). **R** (& **SAS**) language elements and functionality, including computer-science ideas;

(B). Numerical analysis ideas and implementation of statistical algorithms, primarily in **R**; and

(C). Data analysis and statistical applications of (A)-(B).

The object of the course is to reach a point where students have some facility in generating statistically meaningful models and outputs. Wherever possible, the use of **R** and numerical-analysis concepts is illustrated in the context of analysis of real or simulated data. The assigned homework problems will have the same flavor.

The course formerly introduced **Splus**, where now we emphasize the use of **R**. The syntax is very much the same for the two packages, but **R** costs nothing and by now has much greater capabilities. Also, in past terms **SAS** has been introduced primarily in the context of linear and generalized-linear models, to contrast its treatment of those models with the treatment in **R**. Students in this course have often had a separate and more detailed introduction to **SAS** in some other course, so in the present term we will

not present details about **SAS**, in order to leave time for interesting data-analytic topics such as Markov Chain Monte Carlo (`MCMC`) and multi-level modeling in **R**.

Various public datasets will be made available for illustration, homework problems and data analysis projects, as indicated on the course web-page.

The contents of these notes, not all of which are posted currently, and which will be augmented as the term progresses, are:

1. `Introduction to R`
   Unix and R preliminaries, R language basics, inputting data, lists and data-frames, factors, functions.
2. `Random Number Generation & Simulation`
   Pseudo-random number generators, shuffling, goodness of fit testing.
3. `Graphics`
4. `Simulation Speedup Methods`
5. `Numerical Maximization & Root-finding`
   (respectively for log-likelihoods and estimating equations)
6. `Commands for Subsetting`
   Manipulating Arrays and Data Frames
7. `Spline Smoothing Methods`
8. `EM Algorithm`
9. `The Bootstrap Idea`
10. `Markov Chain Monte Carlo`
    Metropolis and Gibbs Sampling Algorithms
    Convergence Diagnostics for MCMC
    Bayesian Data Analysis applications using `WinBugs`
11. `Multi-level Model Data Analysis`
    Linear and Generalized Linear Model Fitting and Interpretation

A few Exercises are contained in these notes, but all formal Homework assignments are posted separately in the course web-page Homework directory.

6 Loose Ends

6.1 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.)

6.2 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.

6.3 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* and *piecewise twice differentiable* function $s : [a, b] \mapsto \mathbf{R}$ 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 $s(x_i) \approx y_i$ when the set of knots becauses smaller, and for a fixed set of knots the solution exhibits more smoothing and less accuracy in satisfying $s(x_i) \approx y_i$ for $x_i \notin \mathcal{K}$ when λ is made larger. The size of `spar` parameter to use is much larger for the function in **R** than it was for the older **Splus** function: values of this parameter of the order of 1 (say from .2 to .6) will be typical. This parameter is proportional to λ above 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` function to see whether you have achieved the desired degree of visual smoothness.

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

```
> x = runif(10)
  y = 4*x^5 - 3*x^2 +2
> tmpspl = smooth.spline(x,y,spar=.3,all.knots=T)
> sort(x)[8:9]
[1] 0.8463247 0.8886385[1] 0.5146857 0.6923524
## Since there are no points between  .85  and .885, let's look at
## the spline-function on that interval !!
>  z = predict(tmpspl, .85+(0:10)*.0035)$y
> var(diff(diff(diff(z))))  ### = 3.2e-30
```

For a vector **z**, `diff(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 $.85, ..8535, \ldots, .885$ is shown by the fact that the third differences are constant.