Avoiding iteration

The canonical bad R program looks like this:

```r
## multiply two vectors
for(i in 1:n){
  d[i]<-a[i]*b[i]
}

## compute the inner product
s<-0
for(i in 1:n){
  s<-s+d[i]
}
```

The right way to do this is:

```r
s<-sum(a*b)
```

Multiplication, like many operations and functions, is vectorized: it works elementwise on whole objects.
Why avoid iteration?

There are two reasons to replace loops with vectorized calculations:

- **Speed**: the `for()` loop is much slower since the expression must be evaluated by the interpreter every time.

- **Clarity**: it is much easier to see what `sum(a*b)` does.

Vectorized functions

These include

- The operators `&`, `|`, `!`, `+`, `-`, `*`, `/`, `^`, `%`.%

- Mathematical functions such as `log`, `sin`, `pnorm`, `choose`, `gamma`, and many more.

- Random number generators such as `rnorm`, `rpois`, ...

- `ifelse` for vectorized conditionals.
Recall the recycling rules:

- If \(a\) and \(b\) are vectors of the same length \(n\), then \(a*b\) is the element-by-element product.
- \(2*b\) should be the vector whose elements are twice those of \(b\), so the 2 must be repeated \(n\) times.
- Generalizing this, the shorter of the two arguments is always repeated to make it as long as the longer argument. If this is not an exact multiple, a warning is given.

---

Matrices and vectors

One of the few examples of vector recycling where the shorter vector isn’t of length 1 is in vector-matrix operations.

```r
> a <- matrix(1:12,nrow=3)
> a
[1,]   1   4   7  10
[2,]   2   5   8  11
[3,]   3   6   9  12

> b <- c(1,10,100)

> a*b
[1,]   1   4   7  10
[2,]  20  50  80 110
[3,] 300 600 900 1200
```
Matrices and vectors

If $a$ is a matrix and $b$ is a vector then $a \times b$ multiplies each column of $a$ by $b$.

This works because a matrix is stored as a vector with the columns stacked on each other. Vector recycling produces a separate copy of $b$ for each column of $a$.

A common application is in regression models that involve computing $WX$, for a design matrix $X$ and diagonal weight matrix $W$.

The code in `lm` and `glm` uses $X \times wts$ where $X$ is the design matrix and $wts$ is the vector diagonal of $W$.

Matrix multiplication

The $\times$ operator performs elementwise multiplication, so another operator $\%\times\%$ is needed for matrix multiplication. Together with $t()$ for matrix transposition this allows many statistical formulas to be written without loops.

$$\hat{\beta} = (X^T WX)^{-1} X^T W Y$$

could be written as

```r
betahat <- solve(t(X) %*% (w * X)) %*% t(X) * (w * Y)
```

However, the QR decomposition approach

```r
whalf <- sqrt(w)
betahat <- solve(whalf * X, whalf * y)
```

is faster, more accurate, and uses less memory!
The main loop construct in R is `for`. The commonest use, as in C and other languages, is to count from 1 to n.

```r
for(i in 1:n){
    ##do something
}
```

But `for()` can iterate over any sequence:

```r
for(i in (1:10)*4)
for(j in c(3,1,4,1,5,9,2,7))
for(variable in names(data))
for(f in c(sin,cos,tan))
```

**Unobvious facts about `for()`**

- If the sequence has length zero, the body of the loop is never executed:
  ```r
  n <- 0
  for(i in seq(length=n)){
      print(i)
  }
  ```
  Note that `i in 1:n` would be wrong here. `1:0` is the sequence `1, 0`. The command `seq(length=n)` should be used defensively in programming.

- In R the loop counter variable exists after the loop finishes:
  ```r
  for(i in 1:10){
      ## do something
  }
  print(i)  ## 10
  ```
  In Splus the variable doesn’t exist after you leave the loop.
Leaving loops

The `break` and `next` commands allow the flow of a loop to be altered:

- **break** jumps out of the loop. For example, in `glm.fit`:
  ```r
  if (abs(dev - devold)/(0.1 + abs(dev)) < control$epsilon) {
      conv <- TRUE
      break
  }
  ```

  jumps out of an iterative optimization when the optimum is (basically) reached.

- **next** jumps to the next iteration of the loop.

Neither of these is very commonly used though.

Other loops

- **while()** repeats an expression while a condition is true. Nearly all the occurrences in base R could equally well use `for()`, but `while` may be used to emphasize that the loop is intended to `break` rather than finish.

- **repeat()** repeats an expression forever, so `break` or an interrupt is needed to terminate it. This is not used at all in base R, but can be useful for demos.
A “why not loop?” example

Suppose \( a \) is a vector of 5 column numbers, indexing a large dataframe \( d \), and you want to compute the mean of each of the indicated columns.

Since `for()` can iterate over anything, you begin

```r
means <- numeric(length(a))
for(i in a){
    means[i]<-mean(d[,i])  ## wrong!
}
```

This doesn’t quite work, because `means` is only 5 entries long, and `i` may be larger than 5.

A “why not loop?” example

Try again, using explicit indexing:

```r
means <- numeric(length(a))
for(i in seq(length=length(a))){
    means[i] <- mean(d[,a[i]])
}
```

This works perfectly well, but is a bit ugly.

A better solution is

```r
means <- sapply(a,function(i) mean(d[,i]))
```

or

```r
means <- apply(d[,a],2,mean)
```
lapply, sapply, apply, tapply all replace loops that iterate over entries or collections of entries in an object, computing the same function on each.

This style of programming is unfamiliar to most people who haven’t used Lisp. It has the advantage of making it clear to the reader (and the computer) that you are doing the same thing to each entry, and that the order of the computations doesn’t matter.

The increased clarity should make the program easier to read and write. In theory it could make it faster, and in fact lapply and sapply are a bit faster than the corresponding loops.

### lapply, sapply

These two are very similar:

- **lapply** \((x, \text{FUN}, \ldots)\) applies FUN to each element of list. Other arguments to FUN can be supplied, and returns the list of results.
- **sapply** works the same way, but simplifies the result to a vector or matrix if possible.

```r
## From anova.lm, ‘objects’ is a list of models
ns <- sapply(objects, function(x) length(x$residuals))
if(any(ns != ns[1]))
  stop("models were not all fitted to the same size of dataset")
```
apply

apply() applies a function to slices of an array (eg columns of a matrix). The syntax is a little tricky: `apply(X, MARGIN, FUN)` where:

- `X` is an array,
- `FUN` is the function to apply,
- `MARGIN` is the dimensions that will be kept, 1 is rows, 2 is columns and so on.

```r
> data(trees)
> apply(trees, 2, mean)
  Girth  Height  Volume
13.24839 76.00000 30.17097
```

returns the mean of each column. This is easy in two dimensions, harder in three.

### 3-d array example

```r
> data(iris3)  ## Anderson’s iris data, in 3-d array
> dimnames(iris3)
[[1]]
NULL
[[2]]
[[3]]
[1] "Setosa"    "Versicolor" "Virginica"
```

```r
> apply(iris3, c(2, 3), mean)  ## keep the variable and species
  Setosa  Versicolor  Virginica
  Sepal L.  5.006  5.936  6.588
  Sepal W.  3.428  2.770  2.974
  Petal L.  1.462  4.260  5.552
  Petal W.  0.246  1.326  2.026
```
tapply applies a function to the cells of an implicit table defined by one or more factors. One example is creating a table from already grouped data.

```r
> data(esoph)
> tapply(esoph$ncases, esoph$agegp, sum)
   25-34 35-44 45-54 55-64 65-74 75+
1      9     46    76    55   13
## could also do xtabs(ncases~alcgp+agegp, data=esoph)

> tapply(esoph$ncases, list(esoph$agegp, esoph$alcgp), sum)
   0-39g/day 40-79 80-119 120+
25-34      0     0     0     1
35-44      1     4     0     4
45-54      1    20    12    13
55-64     12    22    24    18
65-74     11    25    13     6
75+       4     4     2     3
```

**tapply (continued)**

A common data analysis task is to compute tables of means and variances for subsets of the data.

```r
> data(colon)
> tapply(colon$age, list(colon$rx, colon$extent), mean)
   1  2  3  4
Obs 61.0 60.28947 59.58635 55.60000
Lev 53.0 60.25000 60.16988 60.25000
Lev+5FU 55.3 61.90625 59.48606 62.18182

> tapply(colon$age, list(colon$rx, colon$extent),
        function(x) sqrt(var(x)))
   1  2  3  4
Obs 10.745542 13.04742 11.90884 10.565546
Lev 15.257785 10.59730 11.88737  7.531095
Lev+5FU 8.285053 12.84921 12.20495 13.492984
```
The `apply` functions in Splus were at one time substantially faster than the corresponding loops, so people often think of them as a speed optimization.

In fact they may not be much faster in any of the S implementations, and they should really be considered as clarity optimizations.

Optimizing for speed is a separate topic.