Fast(er) R Code

Paul Teetor
Chicago R User's Group
November 2011

Your enemy: "for" loops and data copying.

- A typical R disaster looks like this:

```
for (i in …) {
    for (j in …) {
        dframe <- func(dframe,i,j)
    }
}
```

- *The program spends more time looping and copying than doing useful computation.*

Do It the R Way

- Avoid loops that process one element per iteration
- Use functions that process whole data structures in a single call, eliminating looping and copying
- Vectorized functions: process whole vectors
- The *apply* family of functions: process whole rows, columns, or lists
- Functional programming: ditto

"Vectorized" arithmetic operations can replace the typical `for` loop

- Suppose B and C are vectors
- Instead of explicit element-by-element loop

```
for (i in 1:N) { A[i] <- B[i] + C[i] }
```

  invoke the implicit elem.-by-elem. Operation:

```
A <- B + C
```

- R can use all basic arithmetic operations this way: `A+B`, `A-B`, `A*B`, `A/B`, `A%%B`)
- Example: `sum(B*C)` is the dot product.

Likewise, you can combine a vector and scalar.

- Suppose B is a vector, x is a scalar
- Instead of this explicit element-by-element loop:

```
for (i in 1:N) { A[i] <- B[i] * x }
```

  use the equivalent:

```
A <- B * x
```

- Or B+x or B-x or B/x or B%%x

What if the vectors have unequal lengths?: The Recycling Rule.

Combining a vector and a scalar is just a special case of the Recycling Rule.

B	+	C	=	B	+	C	=	A
10		1		10		1		11
20		2		20		2		22
30		3		30		3		33
40				40		***1***		41
50				50		***2***		52
60				60		***3***		63
70				70		***1***		71
80				80		***2***		82
90				90		***3***		93

Many basic R functions are vectorized: vector in, vector out

- Some functions apply themselves element-by-element to their argument.

```
> A
[1] 1 2 3 4
> sqrt(A)
[1] 1.000000 1.414214 1.732051 2.000000
```

- Similarly for `log(A)`, `exp(A)`, `sin(A)`, etc.
- Key: These functions return vectors, unlike mean(A), median(A) which return scalars

Use 'apply' to calculate functions of rows or columns of a matrix

- apply(*M*, 1, *fun*) = apply *fun* to the rows of *M*
- apply(*M*, 2, *fun*) = apply *fun* to columns of *M*
- *fun* takes one vector argument, returns a vector

```
> M
           [,1]       [,2]       [,3]        [,4]
[1,]  0.7533110  2.0281678 -1.3061853 -0.04203245
[2,]  2.0143547 -2.2168745 -0.8025196  2.15004262
[3,] -0.3551345  0.7583962 -1.7922408 -1.77023084
> apply(M, 1, median)      # median of each row
[1]  0.3556393  0.6059175 -1.0626826
> apply(M, 2, median)      # median of each column
[1]  0.75331105  0.75839618 -1.30618526 -0.04203245
```

Using 'apply': Some Details

- You can use *apply* on data frames by converting to a matrix:

```
apply(as.matrix(dframe), 1, fun)
```

- But be careful: Even one non-numeric column in the data frame causes complete conversion to *character*!
- Cannot use *apply* on a list

lapply: Apply a function to a list

- Suppose *lst* is a list and *fun* is a function
- Then `lapply(`*`lst, fun`*`)` returns a new list:

fun(*lst*[[1]]), *fun*(*lst*[[2]]), *fun*(*lst*[[2]]), . . .

```
> lst <- list(1, 2, 9)
> sqrt(lst)   # sqrt wants a vector, not a list
Error in sqrt(lst) : Non-numeric argument to
mathematical function
> lapply(lst, sqrt)
[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 3
```

The 'apply' family has other members, all with a common theme

- `sapply(`*`lst`*`,`*`fun`*`)` – Like *lapply*, but returns a vector instead of a list
- `mapply(`*`fun`*`,`*lst_1*`,`*lst_2*`,...)` – Apply a function to several lists in parallel
- `vapply(`*`lst`*`,`*`fun`*`,...)` - A faster version of lapply; see the help page for details

Functional programming: The 'Filter' function does searching

- Define a predicate (function), *f*, which is TRUE for the desired elements
- `Filter(`*f*`, `*x*`)` – Returns the elements of *x* for which *f* is true

```
> x
[1] 78 20 98 21 37
> odd <- function(n) (n %% 2 == 1)
> Filter(odd, x)
[1] 21 37
```

Functional Programming: The Reduce function can replace a loop

- `Reduce(f,x)` is a way to iterate over a list or vector, *x*, by applying a function to successive *results* of *f*.

  Suppose *x* = *x1, x2, x2, x4, x5, . . .*

  Then `Reduce(f,x)` successively computes

```
f(x1,x2), x3, x4, . . .
f(f(x1,x2), x3), x4, x5, . . .
f(f(f(x1,x2), x3), x4), x5, . . .
```

Reduce(f,x)

- `f(a,b)` is a function of two arguments, and `x` is a list or vector
- By default, returns value from final *f(...)*
- But you can request the vector of all the intermediate values from *f(...)*
- Useful for iterative calculation that cannot be done with just R's vectorized operations

Toy Examples of Reduce(f,x): sum and cume. product

- Iterative summation:

```
s <- x[1] + x[2]
for (i in 3:length(s)) s <- s + x[i]
```

- Done using Reduce:

```
f <- function(a,b) a + b
s <- Reduce(f, x)
```

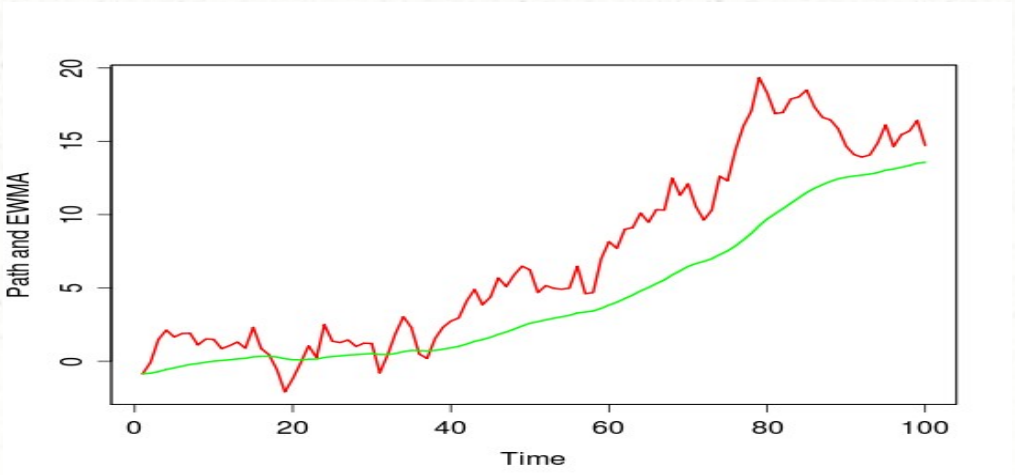
- Cumulative product:

```
f <- function(a,b) a*b
prods <- Reduce(f, x, accumulate=TRUE)
```

Power User Example #1: Exp'ly Weighted Moving Avg.

Suppose price is a vector of daily prices.

```
lambda <- 0.95
f <- function(prv,nxt)
       lambda*prv + (1-lambda)*nxt
ewma <- Reduce(f,prices,accumulate=T)
```



Power User Example #2: Crunching a list of linear models

- *lst* is a list of data frames with *x*, *y* columns
- Transform them into a list of linear models; transform that into a list of slopes:

```
f <- function(df) lm(y ~ x, data=df)
models <- lapply(lst, f)
g <- function(m) coef(m)[2]
slopes <- lapply(models, g)
```

- Try doing *that* in Python!

Fast(er) R Code

- Slides on-line at

  http://quanttrader.info/public

- Code snippets under

  https://github.com/pteetor/public

  . . . in the CRUG-2011-Nov directory

paulteetor@yahoo.com

@pteetor