7. Embedding C code

Thomas Lumley
Ken Rice

Universities of Washington and Auckland

Seattle, July 2013

Why C?

- Some tasks are slow and require explicit loops that can't be vectorized away.

- C is simple and allows for efficient implementation of exactly the kinds of algorithms that R is no good at.

- C is standardized, portable, and has good-quality free compilers on effectively all platforms.

You can also embed C++, Fortran, Java, Perl, Python, or even OCaml

A little example

The convolution of two sequences $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$ is the sum

$$z_i = \sum_{j+k=i} x_j y_k$$

If x and y are probability mass functions, z is the probability mass function of the sum, so this computation is useful in statistical calculations.

A simple R version is

```
m <- length(x)
n <- length(y)
z <- numeric(m+n)
for(j in 1:m){
    for (k in 1:n){
         z[j+k-1] = z[j+k-1] + x[j]*y[k]
    }
}
```

A little example

One loop can be removed easily, but removing both loops doesn't seem to be possible (without using $m \times n$ memory), so convolution is a good candidate for translating to C.

```
void convolve(double *x, int *n, double *y, int *m, double *z){
       int i, j, nz = *m + *n - 1;
       for(i = 0; i < nz; i++) z[i] = 0.0;

       for(i = 0; i < *n; i++) {
         for(j = 0; j < *m; j++){
           z[i + j] += x[i] * y[j];
         }
       }
     }
```

The C code is very similar to the R code, another indication that the R code will be slow.

Notes:

- All arguments are passed as pointers, since all R data types are vectors.

- Lengths of vectors can't be determined in C, so need to be passed in.

- Return type is `void`, so values are returned by modifying arguments

- The arguments are copies of the R objects, not the originals.

Compiling and linking

On Unix or Mac OS, or on Windows with the `Rtools` toolchain, from the OS command line

```
R CMD SHLIB convolve.c
```

to make a dynamic library (`convolve.dll` under Windows, `convolve.so` under most Unix).

In R

```
dyn.load("convolve.so")
```

or

```
dyn.load(paste("convolve", .Platform$dynlib.ext, sep=""))
```

for portability. [Or make a package]

Other compilers

You can make packages with almost any C compiler. On Unix-like systems this typically just works: the C binary interface is standardized.

On Windows there is less standardization and you need the right compiler options. Instructions for some popular ones under Windows. are at http://www.stats.uwo.ca/faculty/murdoch/software/compilingDLLs/

Not all compilers will correctly compile R itself, in particular it is difficult to compile R with the Microsoft C/C++ compiler. R relies heavily on details of the IEEE floating point standard.

Calling from R

```
conv <- function(x, y){
    .C("convolve", x=as.double(x), n=length(x),
        y=as.double(y), m=length(y),
        z=numeric(length(x)+length(y)-1))$z
}
```

Need to make sure the arguments are of the correct type (double or integer), and need to supply an empty vector for the result.

Argument names are ignored by R but help us keep track.

`.C()` returns a list with copies of all the arguments, but we only care about the last argument.

Calling from R

The C code gives the same answers as the R code above, but much faster

```
> system.time(for (i in 1:100) conv(rep(1,100),rep(1,100)))
   user  system elapsed
  0.006   0.000   0.006
> system.time(for (i in 1:100) Rconv(rep(1,100),rep(1,100)))
   user  system elapsed
  8.567   0.018   8.600
```

More realistic example

Given a gene expression value X and a phenotype Y, find the best (smallest p-value) way to divide X into two categories to predict Y.

```
cutpoints <- sort(unique(x))
n<-length(cutpoints)
pvalues <- sapply(cutpoints[3:(n-2)],
     function(c) {
       z <- x<c
       t.test(y~z)$p.value
     })
best <- which.min(pvalues)
cutpoints[3:(n-2)][best]
```

Computing the unique values of X is fast in R; the loop over cutpoints is slow.

More realistic example

Design for C

- Sort X in R first
- Keep sums `1:i`, `(i+1):n` of Y and Y^2, update by adding/subtracting ith term in loop
- Compute squared z statistic rather than p-value.

More realistic example

```
void bestz(double x[], double y[], int *n, int *best){
   double sum1 = 0, sum2 = 0,  sumsq1 = 0, sumsq2 = 0;
   double mean1, mean2, var1, var2;
   double best_zsq = -1, zsq;
   int N = *n;
   int i;

    for(i=2; i<N; i++){
      sum1 += y[i];
      sumsq1 += y[i]*y[i];
    }
    sum2=y[0]+y[1];
    sumsq2=y[0]*y[0]+y[1]*y[1];
    *best = -1;
```

More realistic example

```
    for(i=2; i<N-1;  i++){
      mean1 = sum1/(N-i);
      mean2 = sum2/i;
      var1 = (sumsq1/(N-i))- mean1*mean1;
      var2 = (sumsq2/i) - mean2*mean2;
      zsq= (mean1-mean2)*(mean1-mean2)/(var1/(N-i)+var2/i);
      if (zsq>best_zsq) {
        *best=i;
         best_zsq=zsq;
      }
      sum1 -= y[i];
      sum2 += y[i];
      sumsq1 -= y[i]*y[i];
      sumsq2 += y[i]*y[i];
    }  /* i */
}      /* function */
```

More realistic example

From R

```
dyn.load("bestz.so")

bestz <- function(x,y){
   i <- order(x)
   n <- length(x)
   if (length(y)!=n) stop("lengths don't agree")
   best <- .C("bestz", x=as.double(x[i]), y=as.double(y[i]),
          n=n, best=integer(1))$best
   ibest <- i[best]+1
   list(cutpoint= x[ibest], test= t.test(x<x[ibest], y))
}
```

More realistic example

C code is much faster, for two reasons

- In C

- C code is $O(n)$, R code is $O(n^2)$ because it recomputes the means and variances from scratch.

Note: If $Y|X$ has constant variance, an even faster pure-R approach, based on changepoint theory, is

```
i<-order(x)
which.max( cumsum(y[i]-mean(y)))
```

C in packages

Put C code in the `src/` subdirectory of your package. It will be compiled and linked automatically when the package is installed.

Put `useDynLib(pkgname)` in the NAMESPACE file to load `pkgname.dll` (or `pkgname.so` or whatever).

Calls to `.C` from code in your package will now automatically look only in `pkgname.dll` for compiled routines.

.Call()

The `.C` interface is useful only for arithmetic, logical, and string vectors.

Calling back to R is clumsy and handling more complicated R objects such as lists is not feasible.

Most error checking must be done in R as it cannot be done in C and type or length errors will corrupt memory.

`.Call` provides an alternative interface that passes pointers to R objects and returns an R object.

.Call and convolve

```
#include "Rinternals.h"

SEXP  convolve(SEXP x, SEXP y){
       int i, j, m,n, nz;
       SEXP z;

       m = LENGTH(x);
       n = LENGTH(y);

       PROTECT(z = allocVector(REALSXP, m+n-1));
       for(i =0; i< n+m-1; i++) REAL(z)[i]=0;

       for(i = 0; i < m; i++) {
         for(j = 0; j < n; j++){
           REAL(z)[i + j] += REAL(x)[i] * REAL(y)[j];
         }
       }
       UNPROTECT(1);  /*z*/
       return z;
     }
```

Notes

- `SEXP`, short for S-expression (from LISP) is the type of R objects.
- `LENGTH()` returns the length of vector
- `REAL()` is a pointer to the actual numbers in the R object (`INTEGER`, `LOGICAL` for other types). `REALSXP` indicates the numeric type.
- `PROTECT()` protects memory from the garbage collector, `UNPROTECT()` releases it.
- Pointer protection is a stack: need to match `PROTECT` and `UNPROTECT` calls. The returned value is Someone Else's Problem.

Improvements

Computing the vector lengths in C removes one source of errors.

`allocVector()` will give an R-level error if memory is not available.

The `REAL()` function will give an R-level error (rather than corrupting memory) if the arguments are not numeric.

Still better to check explicitly and to convert integer or logical arguments to numeric.

Also, there is some overhead to calling `REAL()` each time.

Improvements

```
SEXP xconv,yconv;
double *xdata, *ydata;
/*...*/

if (TYPEOF(x)==REALSXP){
  xdata = REAL(x);
  xconv = 0;
} else {
  xconv = coerceVector(x, REALSXP);
  xdata= REAL(xconv);
}
/* ... */

     for(i = 0; i < n; i++) {
         for(j = 0; j < m; j++){
           zdata[i+j] += xdata[i] * ydata[i];
         }
       }
/*...*/

if (yconv) UNPROTECT(1);
if (xconv) UNPROTECT(1);
UNPROTECT(1)  /* z */
```

Lists, functions, expressions

A stripped-down version of `lapply()` (used in deciding whether to move `lapply()` to C). Takes an expression in `x` rather than a function.

```
#include "Rinternals.h"
SEXP elapply(SEXP list, SEXP expr, SEXP rho)
     {
       R_len_t i, n = length(list);
       SEXP ans;

       if(!isNewList(list)) error("'list' must be a list");
       if(!isEnvironment(rho)) error("'rho' should be an environment");
       PROTECT(ans = allocVector(VECSXP, n));
       for(i = 0; i < n; i++) {
         defineVar(install("x"), VECTOR_ELT(list, i), rho);
         SET_VECTOR_ELT(ans, i, eval(expr, rho));
       }
       setAttrib(ans, R_NamesSymbol, getAttrib(list, R_NamesSymbol));
       UNPROTECT(1);
       return ans;
     }
```

Translation

For each element of the list in turn

- Set `x` to the `i`th element of the list
- evaluate `expr` with that value of x
- put the value in the `i`th element of the answer

```
l <- list(1, 2, 3, 75)
.Call("elapply", l, quote(x^2), new.env())
```

Notes

- `isNewList()` checks for a list, `isEnvironment()` checks for an environment.

- `defineVar()` assigns a value to a variable, `install("x")` puts `x` in R's symbol table and returns a code.

- `VECTOR_ELT` reads an element from a list, `SET_VECTOR_ELT` writes an element to a list

- `eval` evaluates an expression

- `getAttrib()` and `setAttrib()` read and set attributes.

`inline` package

Allows C (C++, Fortran, Objective-C) code to be written in-line in R code, as a character string or vector.

`cfunction()` writes the C function declaration, includes necessary header files, compiles the code, and writes an R function that uses `.C()` or `.Call()`.

Example: `.Call()` version of `convolve`

inline package

```
convinline <- cfunction(
    sig=signature(x="numeric",
                  y="numeric"),
    body="
       int i, j, m,n, nz;
       SEXP z;

       m = LENGTH(x);
       n = LENGTH(y);

       PROTECT(z = allocVector(REALSXP, m+n-1));
       for(i =0; i< n+m-1; i++) REAL(z)[i]=0;

       for(i = 0; i < m; i++) {
         for(j = 0; j < n; j++){
           REAL(z)[i + j] += REAL(x)[i] * REAL(y)[j];
         }
       }
       UNPROTECT(1);  /*z*/
       return z;
      ",
    convention=".Call", language="C")
```

`inline` package

produces an S4 object inheriting from `function`

```
> convinline@.Data
function (x, y)
{
    if (!file.exists(libLFile))
        libLFile <<- compileCode(f, code, language, verbose)
    if (!(f %in% names(getLoadedDLLs())))
        dyn.load(libLFile)
    .Call("file3c7812be", PACKAGE = f, x, y)
}
```

inline package

The C code has been wrapped up into a full program:

```
> cat(convinline@code)
#include <R.h>
#include <Rdefines.h>
#include <R_ext/Error.h>


SEXP file6f1696f5 ( SEXP x, SEXP y ) {

        int i, j, m,n, nz;
        SEXP z;

        m = LENGTH(x);
        n = LENGTH(y);

        PROTECT(z = allocVector(REALSXP, m+n-1));
        for(i =0; i< n+m-1; i++) REAL(z)[i]=0;
    [...snip...]
        UNPROTECT(1);  /*z*/
        return z;

  warning("your C program does not return anything!");
  return R_NilValue;
}
```

Debugging

You can run R under a debugger, such as `gdb`

```
R  --debugger="gdb"
```

Type `run` to run R, then load the compiled code, then `CTRL-C` to get back to the debugger.

Set break points with eg

```
break elapply
break elapply.c:22
```

Valgrind

Under Linux, `valgrind` is a memory access checker that runs code in a virtual machine. It catches many typical C errors such as reading or writing off the end of an array.

```
==12539== Invalid read of size 4
==12539==    at 0x1CDF6CBE: csc_compTr (Mutils.c:273)
==12539==    by 0x1CE07E1E: tsc_transpose (dtCMatrix.c:25)
==12539==    by 0x80A67A7: do_dotcall (dotcode.c:858)
==12539==    by 0x80CACE2: Rf_eval (eval.c:400)
==12539==    by 0x80CB5AF: R_execClosure (eval.c:658)
==12539==    by 0x80CB98E: R_execMethod (eval.c:760)
==12539==    by 0x1B93DEFA: R_standardGeneric (methods_list_dispatch.c:624)
==12539==    by 0x810262E: do_standardGeneric (objects.c:1012)
==12539==    by 0x80CAD23: Rf_eval (eval.c:403)
==12539==    by 0x80CB2F0: Rf_applyClosure (eval.c:573)
==12539==    by 0x80CADCC: Rf_eval (eval.c:414)
==12539==    by 0x80CAA03: Rf_eval (eval.c:362)
==12539==  Address 0x1C0D2EA8 is 280 bytes inside a block of size 1996 alloc'
==12539==    at 0x1B9008D1: malloc (vg_replace_malloc.c:149)
==12539==    by 0x80F1B34: GetNewPage (memory.c:610)
==12539==    by 0x80F7515: Rf_allocVector (memory.c:1915)
```

Other C resources

- *Writing R Extensions* manual
- *The C Programming Language* Kernighan & Ritchie (2nd edition)
- Steve Summit's notes at http://www.eskimo.com/~scs/cclass/