Why use externally routines?

- One of the design objectives of the S language was to use existing code where possible.
- High-quality, well-tested code is available for important applications like numerical linear algebra.
- Some calculations are clumsy to express in S and can be more efficiently implemented in compiled languages. Note, however, that it is common to assume that some part of the code is using most of the execution time when, in fact, it is not. Always check by profiling the code before optimizing.

Using .Fortran

Suppose we want to compute the LU decomposition of a matrix. The Lapack subroutine to do this is declared as:

```fortran
SUBROUTINE DGETRF( M, N, A, LDA, IPIV, INFO )
*   .. Scalar Arguments ..
INTEGER INFO, LDA, M, N
*   ..
*   .. Array Arguments ..
INTEGER IPIV( * )
DOUBLE PRECISION A( LDA, * )
```
To call `dgetrf` from R we use `.Fortran`.

```r
> mm = matrix(rnorm(16), nr = 4)
> mmdc = .Fortran("dgetrf", m = as.integer(4), n = as.integer(4),
+ a = as.double(mm), lda = as.integer(4), ipiv = integer(4),
+ info = integer(1))
> str(mmdc)
```

```
List of 6
$ m : int 4
$ n : int 4
$ a : num [1:16] 1.391 0.141 -0.281 -0.106 -1.375 ... 
$ lda : int 4
$ ipiv: int [1:4] 4 3 4 4
$ info: int 0
```

It is common to write a short “wrapper” function in R to marshall the arguments and to check the results.

```r
> LUdecom = function(x) {
+ x = as.matrix(x)
+ m = nrow(x)
+ n = ncol(x)
+ storage.mode(x) = "double"
+ dc = .Fortran("dgetrf", m = m, n = n, a = x,
+ lda = m, ipiv = integer(min(m, n)), info = integer(1))
+ if (dc$info)
+ stop(paste("dgetrf returned error code",
+ dc$info))
+ list(lu = dc$a, ipiv = dc$ipiv)
+ }
> mmdc = LUdecom(mm)
> str(mmdc)
```

```
List of 2
$ lu : num [1:4, 1:4] 1.391 0.141 -0.281 -0.106 -1.375 ... 
$ ipiv: int [1:4] 4 3 4 4
```
Using .C

- The R function .C is used to call a C function that returns void.
- All arguments to the C function must be of type int * or double *
  or, less commonly, Rcomplex * or char **.
- We give an example from the manual "Writing R Extensions", which you should read for more details.
- The example is the calculation of the convolution of two vectors, an operation that would be difficult to phrase in S without time-consuming explicit looping.
- The “output” object must be created in R and passed to the .C call.

Using .C (cont’d)

The C function is defined as

```c
void convolve(double *a, int *na, double *b, int *nb, double *ab) {
    int i, j, nab = *na + *nb - 1;
    for(i = 0; i < nab; i++)
        ab[i] = 0.0;
    for(i = 0; i < *na; i++)
        for(j = 0; j < *nb; j++)
            ab[i + j] += a[i] * b[j];
}
```

and the corresponding function in R is

```r
> conv <- function(a, b) .C("convolve", as.double(a),
+ as.integer(length(a)), as.double(b), as.integer(length(b)),
+ ab = double(length(a) + length(b) - 1))$ab
```

Using .Call

- Using .C is fine if you are doing a strictly numerical calculation but it can get clumsy if you are manipulating structures or classed objects.
- Most of R is written in C hence the internal representations of R objects are accessible to the C programmer.
- The .Call interface passes raw R objects and returns a raw R object.
- The .Call interface is very powerful when used carefully. When used clumsily it is a fast way of acquainting yourself with the meaning of the phrase “Segmentation fault”.
- R provides garbage collection. This is both a blessing and a curse. When creating R objects in C code you must account for the possibility of a garbage collection occurring and protect the objects you create.
- Before returning from a function, you must unprotect the same number of pointers that you have protected.
- It is a good idea to include the header file Rdefines.h and use the macros available there.

Using .Call (cont’d)

```c
#include <Rdefines.h>
SEXP convolve2(SEXP a, SEXP b) {
    int i, j, na, nb, nab;
    double *xa, *xb, *xab;
    SEXP ab;

    PROTECT(a = AS_NUMERIC(a));
    PROTECT(b = AS_NUMERIC(b));
    na = LENGTH(a); nb = LENGTH(b); nab = na + nb - 1;
    PROTECT(ab = NEW_NUMERIC(nab));
    xa = NUMERIC_POINTER(a); xb = NUMERIC_POINTER(b);
    xab = NUMERIC_POINTER(ab);
    for(i = 0; i < nab; i++) xab[i] = 0.0;
    for(i = 0; i < na; i++)
        for(j = 0; j < nb; j++)
            xab[i + j] += xa[i] * xb[j];
    UNPROTECT(3);
    return(ab);
}
```

```r
> convolve = function(a, b) .Call("convolve2", a, b)
```
Using `.Call (cont’d)

- The R code is now much more straightforward
- All arguments to the C function and its value are defined to be of type SEXP (a pointer to a symbolic expression - the name is a holdover from LISP).
- Unless you can control how the C function will be called, you should expect to test and possibly modify the storage mode of your arguments. The macro AS_NUMERIC is similar to the R function as.double. Because it may copy the contents of its argument to a new R object with a different storage mode, you must PROTECT the result.

Manipulating R classed objects

- The combination of S4 classes and the `.Call` interface is particularly powerful. Because the types and names of slots in a classed object are defined in a class definition and enforced by the methods package, you can access them with confidence.
- Macros GET_SLOT, SET_SLOT, NEW_OBJECT and MAKE_CLASS allow interaction with classed objects.

LU decomposition class and constructor

```r
> setClass("LUdecomposition", representation(a = "matrix", + pivot = "integer"))
> setClass("dmatrix", "matrix", validity = function(object) mode(object) == + "numeric")
> setClass("DGEmatrix", "dmatrix")
> setGeneric("decompose", function(object, ...) standardGeneric("decompose"))
> setMethod("decompose", "DGEmatrix", function(object, + ...) .Call("La_DGE_dc", object))
```

C code for LU decomposition

```c
SEXP La_DGE_dc(SEXP A)
{
    SEXP aa = PROTECT(duplicate(A));
    SEXP adims, pivot, val;
    int m, n, info;

    if (!isMatrix(aa) || !isReal(aa)) {
        error("A must be a double precision matrix");
    }

    adims = GET_DIM(aa); m = INTEGER(adims)[0]; n = INTEGER(adims)[1];
    pivot = PROTECT(NEW_INTEGER(m < n ? m : n));
    F77_CALL(dgetrf)(&m, &n, REAL(aa), &m, INTEGER(pivot), &info);
    check_Lapack_error(info, "dtrtrf");
    val = PROTECT(NEW_OBJECT(MAKE_CLASS("LUdecomposition")));
    SET_SLOT(val, install("a"), aa);
    SET_SLOT(val, install("pivot"), pivot);
    UNPROTECT(3);
    return val;
}
```
**LU decomposition**

- The value of the argument will be modified in the Fortran routine. It must be duplicated before you manipulate it.
- Macros such as `GET_DIM` and functions like `isMatrix` are handy for getting and testing R characteristics in C.
- The `SET_SLOT` macro (and its complement `GET_SLOT`) are used with classed objects. The second argument to these macros must be an R object of mode name. The `install` function converts a C character string to a name. It must always be used (but it is very fast because it accesses a hash table).
- The next example shows using a LUdecomposition object to calculate the inverse of the original matrix.

**Passing classed objects**

```c
SEXP La_DGE_tri(SEXP LU)
{
    SEXP aa = PROTECT(duplicate(GET_SLOT(LU, install("a"))));
    SEXP pivot = GET_SLOT(LU, install("pivot"));
    SEXP adims = GET_DIM(aa);
    int m = INTEGER(adims)[0], n = INTEGER(adims)[1];
    int info, lwork = -1;
    double tmp, *work;

    if (m != n) {
        error("LU must be the LU decomposition of a square matrix");
    } /* first call to dgetri gets optimal size of work array */
    F77_CALL(dgetri)(&n,REAL(aa),&n,INTEGER(pivot),&tmp,&lwork,&info);
    check_Lapack_error(info, "dtrtri");
    lwork = (int) tmp;
    work = Calloc(lwork, double);
    F77_CALL(dgetri)(&n,REAL(aa),&n,INTEGER(pivot),work,&lwork,&info);
    check_Lapack_error(info, "dtrtri");
    Free(work);
    UNPROTECT(1);
    return aa;
}
```

**Comments on La_DGE_tri**

- We extract the slots we need from the classed object.
- We again duplicate the matrix before changing it.
- The first call to the Lapack routine uses the value \(-1\) for `lwork`. The Lapack convention is to calculate the optimal value of `lwork` and pass that back as the first element of the argument `work`. We use `Calloc` to allocate the work array and `Free` to free it when it is no longer needed.
- The C function `check_Lapack_error` checks if the error code is non-zero and, if so, prints an informative error message in a call to `error`.

```c
void check_Lapack_error(const int info, const char* name)
{
    if (info == 0) return;
    if (info < 0) error("Argument %d to Lapack function %s is illegal",
                        -info, name);
    error("Lapack function %s returned error code %d", name, info);
}
```