

Topics

- we are going to see:
 - how to allocate a new matrix, set values to its cells and return it
 - how to use the `runif()`, `rnorm()`, ..., the C functions underlying R could be called from C using `#include <Rmath.h>`
 - how to pass a matrix to a C function and modify it within the function
 - how to pass an array to a C function and modify it within the function

Return A Matrix I

- allocate a new matrix: (within R, a matrix is really a long vector, i.e. it is stored in the so-called “vec” form)

```
SEXP ans;  
PROTECT(ans = allocVector(REALSXP, nrow * ncol));  
++nProtected;
```

- set the (ii, jj) -th element of a matrix:

```
double value;  
REAL(ans)[ii + nrow * jj] = value;
```

- setting its dimensions right, so far we have a vector really!

```
SEXP dim;  
PROTECT(dim = allocVector(INTSXP, 2));  
++nProtected;  
INTEGER(dim)[0] = nrow; INTEGER(dim)[1] = ncol;  
setAttrib(ans, R_DimSymbol, dim);
```

- return a matrix:

```
UNPROTECT(nProtected);  
return(ans);
```

Using Rmath.h

- the include statement:

```
#include <Rmath.h>
```

- before calling *any* random number generator function call:

```
GetRNGstate();
```

- after having done with *all* the random number generator functions, call:

```
PutRNGstate();
```

- use the `rnorm()` function:

```
double mu, sigma;  
REAL(ans)[ii + nrow * jj] = rnorm(mu, sigma);
```

- take a look at the file `R-2.1.0/include/Rmath.h` to discover lots of useful distribution related functions

An Example I

- goal:

```
/*
 * Given two vectors mean and sd of length mm each, this function
 * returns a (nobs \times mm) matrix whose nobs numbers ii-th column
 * is a random sample from Normal(mean[ii], sd[ii]^2)
 */
```

- the R side:

```
getRnormMatrix <-
  function (mean, sd, nobs)
  {
    stopifnot(is.vector(mean)); stopifnot(is.vector(sd))
    stopifnot(all(sd > 0)); stopifnot(nobs > 0)
    mat <- .Call("get_rnorm_matrix",
                 as.numeric(mean), as.numeric(sd), as.integer(nobs))
    mm <- length(mean)
    dimnames(mat) <- list(paste("obs", 1:nobs, sep = ""),
                         paste("distrib", 1:mm, sep = ""))
    return(mat)
  }
```

An Example II

- the C side (part I):

```
SEXP
get_rnorm_matrix (SEXP mean, SEXP sd, SEXP nobs)
{
    int nProtected = 0;
    R_len_t ii, jj, ncol, nrow;
    double mu, sigma;
    SEXP ans, dim;

    ncol = length(mean);
    if (ncol != length(sd)) {
        error("the vectors mean and sd should be of the same length");
    }
    nrow = INTEGER(nobs)[0];

    PROTECT(ans = allocVector(REALSXP, nrow * ncol));
    ++nProtected;
```

An Example III

- the C side (part II):

```
GetRNGstate();
for (jj = 0; jj < ncol; ++jj) {
    mu = REAL(mean)[jj];
    sigma = REAL(sd)[jj];
    for (ii = 0; ii < nrow; ++ii) {
        REAL(ans)[ii + nrow * jj] = rnorm(mu, sigma);
    }
}
PutRNGstate();

PROTECT(dim = allocVector(INTSXP, 2));
++nProtected;
INTEGER(dim)[0] = nrow; INTEGER(dim)[1] = ncol;
setAttrib(ans, R_DimSymbol, dim);

UNPROTECT(nProtected);
return(ans);
}
```

Pass A Matrix I

- passing a matrix as a vector and its dimensions:
 - we have a function `foo1(SEXP mat, SEXP dims)` taking a matrix as a vector and its dimensions
 - within `foo1()`, we use `mat` and `dims` as follows

```
R_len_t ii, jj, nrow, ncol;  
double mu;  
nrow = INTEGER(dims)[0];  
ncol = INTEGER(dims)[1];  
mu = REAL(mat)[ii + ncol * jj];
```

Pass A Matrix II

- passing a matrix as a matrix:
 - we have a function `foo2(SEXP mat)` taking a matrix
 - within `foo2()`, we use `mat`

```
double mu, sigma;  
R_len_t ii, jj, nrow, ncol;  
nrow = nrows(mat);  
ncol = ncols(mat);  
mu = REAL(mat)[ii + ncol * jj];
```

Example I

- goal:

```
/*  
 * Given a positive number sd and a (nobs \times mm) matrix mat we  
 * return a (nobs \times mm) matrix ret_mat where:  
 *  
 * ret_mat[ii][jj] = mat[ii][jj] + Normal(0, sd^2)  
 */
```

- the R side (pass a matrix as a vector):

```
perturbMatrixByRnorm1 <-  
  function (mat, sd)  
  {  
    stopifnot(is.matrix(mat))  
    stopifnot(sd > 0)  
    mat <- .Call("perturb_matrix_by_rnorm1",  
                as.numeric(mat),  
                as.integer(dim(mat)),  
                as.numeric(sd))  
    return(mat)  
  }
```

Example II

- the C side (part I):

```
SEXP
perturb_matrix_by_rnorm1 (SEXP mat, SEXP dims, SEXP sd)
{
    int nProtected = 0;
    R_len_t ii, jj, nrow, ncol;
    double mu, sigma;
    SEXP ret_mat, dim;

    nrow = INTEGER(dims)[0];
    ncol = INTEGER(dims)[1];

    PROTECT(ret_mat = allocVector(REALSXP, nrow * ncol));
    ++nProtected;
```

Example III

- the C side (part II):

```
GetRNGstate();
sigma = REAL(sd)[0];
for (ii = 0; ii < nrow; ++ii) {
    for (jj = 0; jj < ncol; ++jj) {
        mu = REAL(mat)[ii + nrow * jj];
        REAL(ret_mat)[ii + nrow * jj] = rnorm(mu, sigma);
    }
}
PutRNGstate();

PROTECT(dim = allocVector(INTSXP, 2));
++nProtected;
INTEGER(dim)[0] = nrow; INTEGER(dim)[1] = ncol;
setAttrib(ret_mat, R_DimSymbol, dim);

UNPROTECT(nProtected);
return(ret_mat);
}
```

Matrix Allocation

- one can avoid first allocating a vector and then setting its dim, i.e., things like the following:

```
PROTECT(ret_mat = allocVector(REALSXP, nrow * ncol));  
++nProtected;
```

```
PROTECT(dim = allocVector(INTSXP, 2));  
++nProtected;  
INTEGER(dim)[0] = nrow; INTEGER(dim)[1] = ncol;  
setAttrib(ret_mat, R_DimSymbol, dim);
```

- the way to avoid this:

```
PROTECT(ret_mat = allocMatrix(REALSXP, nrow, ncol));  
++nProtected;
```

Example I

- goal (same as before):

```
/*
 * Given a positive number sd and a (nobs \times mm) matrix mat we
 * return a (nobs \times mm) matrix ret_mat where:
 *
 * ret_mat[ii][jj] = mat[ii][jj] + Normal(0, sd^2)
 */
```

- the R side:

```
perturbMatrixByRnorm2 <-
  function (mat, sd)
  {
    stopifnot(is.matrix(mat))
    stopifnot(sd > 0)
    mat <- .Call("perturb_matrix_by_rnorm2",
                 as.matrix(mat),
                 as.numeric(sd))
    return(mat)
  }
```

Example II

- the C side (part I):

```
SEXP
perturb_matrix_by_rnorm2 (SEXP mat, SEXP sd)
{
    int nProtected = 0;
    R_len_t ii, jj, nrow, ncol;
    double mu, sigma;
    SEXP ret_mat;

    nrow = nrows(mat);
    ncol = ncols(mat);

    PROTECT(ret_mat = allocMatrix(REALSXP, nrow, ncol));
    ++nProtected;
```

Example III

- the C side (part II):

```
GetRNGstate();
sigma = REAL(sd)[0];
for (ii = 0; ii < nrow; ++ii) {
    for (jj = 0; jj < ncol; ++jj) {
        mu = REAL(mat)[ii + nrow * jj];
        REAL(ret_mat)[ii + nrow * jj] = rnorm(mu, sigma);
    }
}
PutRNGstate();

UNPROTECT(nProtected);
return(ret_mat);
}
```

Pros and Cons

- pros and cons:
 - while passing as vector dimensions have to be passed separately
 - passing as matrix doesn't require extra dimension passing, one could use `nrows()` and `ncols()`
 - `allocVector` works for vectors, matrices and arrays
 - `allocMatrix` only works for matrices
 - `allocVector` does require the dimension setting bits:
`setAttrib(ret_arr, R_DimSymbol, dim);`
 - `allocMatrix` doesn't require the dimension setting bits:
`setAttrib(ret_arr, R_DimSymbol, dim);`

Handling Arrays I

- recap: matrices within R are stored in “vec” form, e.g. if `mat` is a matrix of dimensions `c(dim1, dim2)` then its stored as
`c(mat[, 1], mat[, 2], ..., mat[, ndim2])`
- arrays (lets consider only three-dimensional arrays) are stored in their “vec” form, e.g if `arr` is an array of dimensions `c(dim1, dim2, dim3)` then its stored as
`c(c(arr[, , 1]), c(arr[, , 2]), ..., c(arr[, , ndim3]))`
- note e.g. `arr[, , 1]` is a matrix and hence `c(arr[, , 1])` in above means the “vec” form of the matrix `arr[, , 1]`

Handling Arrays II

- goal:

```
/*  
 * Given an array arr and its dimension dims this function returns a  
 * new array of same dimension such that:  
 *  
 * ret_arr[ii][[jj][kk] = arr[ii][[jj][kk] * arr[ii][[jj][kk]  
 */
```

- the R side:

```
squareArray <-  
  function (arr)  
  {  
    stopifnot(is.array(arr))  
    .Call("square_array",  
          as.numeric(arr),  
          as.integer(dim(arr)))  
  }
```

Example I

- the C side (part I): (you should be able to read and make sense of this code if you understood the code for matrices appearing earlier)

```
#define N_ARR_DIM 3

SEXP
square_array (SEXP arr, SEXP dims)
{
    int nProtected = 0;
    R_len_t ii, jj, kk, ll, dd[N_ARR_DIM], nn = 1;
    double tmp;
    SEXP ret_arr, dim;

    for (ii = 0; ii < N_ARR_DIM; ++ii) {
        dd[ii] = INTEGER(dims)[ii];
        nn *= dd[ii];
    }

    PROTECT(ret_arr = allocVector(REALSXP, nn));
    ++nProtected;
```

Example II

- the C side (part II):

```
    for (kk = 0; kk < dd[2]; ++kk) {
        for (ii = 0; ii < dd[0]; ++ii) {
            for (jj = 0; jj < dd[1]; ++jj) {
                ll = (kk * dd[0] * dd[1]) + (ii * dd[1]) + jj;
                tmp = REAL(arr)[ll];
                REAL(ret_arr)[ll] = tmp * tmp;
            }
        }
    }

    PROTECT(dim = allocVector(INTSXP, N_ARR_DIM));
    ++nProtected;
    for (ii = 0; ii < N_ARR_DIM; ++ii)
        INTEGER(dim)[ii] = dd[ii];
    setAttrib(ret_arr, R_DimSymbol, dim);

    UNPROTECT(nProtected);
    return ret_arr;
}
```

Code Files

prog5.c

prog5.R

prog6.c

prog6.R