Arrays from R to C and Back

Jan de Leeuw

First created on May 21, 2020. Last update on June 02, 2020

Abstract

For multidimensional arrays and symmetric arrays in compact storage the mapping of R array indices to C storage locations and its inverse can be complicated. This short note discusses and implements a systematic way of handling this mapping for vectors, general matrices, symmetric matrices in compact storage, general multidimensional arrays, and super-symmetric multidimensional arrays.

Contents

Note: This is a working paper which will be expanded/updated frequently. All suggestions for improvement are welcome. The directory deleeuwpdx.net/pubfolders/array has a pdf version, the bib file, the complete Rmd file with the code chunks, and the R and C source code.

1 Introduction

Arrays in R are vectors with a dim attribute. Arrays in C are blocks of consecutive memory, and references to arrays *usually* decay to pointers to the initial element of the block. Arrays of pointers can be used to create, or maybe we should say simulate, multidimensional arrays.

There are two main differences between arrays in R and C. The first is that array indexing in R starts with 1, and in C it starts with 0. Thus in an R array x the first element is $x[1]$, while in C it is $x[0]$ or, using pointers, $*x$ or $*(x+0)$. The second difference is that in R column-major ordering is used to store the content of arrays of rank two or higher, while C uses row-major storage. Thus if x is the matrix

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

then for the third element in R storage we have $x[3] = x[1,2] = 3$, while in C for the third element we have $*(x+2) = x[2][1] = 2$.

If we pass arrays from R to C (or back) then this can become annoying, especially for arrays of high rank. Even more complications arise if we use compact storage of symmetric arrays, because the mapping of, say, R array indices to C storage locations and its inverse are bound to be more complicated. This short note discusses one systematic way to easily handle that mapping in some of the more important cases.

2 Arrays in C

In array.h we define an array as a C structure with the following elements:

- content - pointer to data of type CTYPE (defaults to double);
- length - pointer to the number of elements in content;
- rank - pointer to number of dimensions of array;
- shape - pointer to vector with number of levels in each dimension;
- encode - pointer to a mapping of C memory locations to R array indices;
- decode - pointer to a mapping of R array indices to C memory locations.

CTYPE is a macro which you can change in array.h from double to other types. encode and decode are of type ENCODE and DECODE, which are typedefs for function pointers.

```
typedef void (*ENCODE)(
    const int *, int *, const int *,
    const int *); // encode gives R array indices from C memory location
typedef void (*DECODE)(
    const int *, int *, const int *,
    const int *); // decode gives C memory location from R array indices
```

For symmetric matrices and arrays shape can be a pointer to a single number.

3 Subroutines

In array.c there are some auxiliary subroutines for sorting integers (using the system qsort()) and for binomial coefficients. The main routines implement encode and decode for arrays with element of type CTYPE in the following classes.

- general vectors (GVENCODE() and GVDECODE())
- general matrices (GMENCODE() and GMDECODE())
- symmetric matrices in compact storage (SMENCODE() and SMDECODE())

- general multidimensional arrays (GAENCODE() and GADECODE())
- super-symmetric multidimensional arrays in compact storage (SAENCODE() and SADECODE())

Compact storage, as used here, is column-major and increasing (for details, see De Leeuw (2017)). Thus array element $x_{i_1 \cdots i_r}$ is stored if and only if $i_1 \leq \cdots \leq i_r$. The symmetric matrix versions of decode and encode can also be used for triangular matrices in compact storage. Of course decode and encode for vectors merely amounts to subtracting or adding one to the argument, so you want not want the overhead of a function call in that case.

All $5 \times 2 = 10$ routines have the same arguments, as in the general prototypes.

```
XXDECODE(const int *indices, int *location, const int *shape, const int *rank);
XXENCODE(int *indices, const int *location, const int *shape, const int *rank);
```

Note that all arguments are pointers to integers. Not all subroutines need both shape and rank, but we always include them to avoid having to deal with a variable number of arguments. Note that we only map from R indices to C storage locations and back, not from C storage locations to R storage locations, or from C indices to R storage locations. Thus the workflow we have in mind is creating our data in R, passing it to C, computing in C using the subroutines, and passing the result back to R. The 10 subroutines are all written so they can be used directly using the .C() interface in R (i.e. they pass all arguments by reference, return void, and have no I/O). The main reason for not using the .Call() interface is that we want to make the routines usable for people who do not have R. We may try at some point to change the R code to use the .C64() interface (Gerber, Mösinger, and Furrer (2018)).

Using the array structure with pointers to corresponding subroutines will make it easy to extend our results to other types of arrays, such as triangular and banded matrices, and even to Hilbert, Vandermonde, Toeplitz, or Hankel matrices.

4 Testers

I have also included five main programs in C which test the decode and encode routines. They are named appropriately:

- gvtester();
- gmtester();
- smtester();
- gatester();
- satester().

5 Appendix: Code

5.1 array.h

```
#ifndef INDEXING_H
#define INDEXING_H

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define CTYPE double

typedef void (*ENCODE)(
    const int *, int *, const int *,
    const int *); // encode gives R array indices from C memory location
typedef void (*DECODE)(
    const int *, int *, const int *,
    const int *); // decode gives C memory location from R array indices

typedef struct {
  CTYPE *content;
  int *length;
  int *rank;
  int *shape;
  ENCODE encode;
  DECODE decode;
} ARRAY;

/* protoypes */

extern int binCoef(const int, const int);

extern void isort(int *, const int *);

extern int icmp(const void *, const void *);

extern inline int IMIN(const int, const int);

extern void GVENCODE(const int *, int *, const int *, const int *);

extern void GVDECODE(const int *, int *, const int *, const int *);
```

```
extern void GMDECODE(const int *, int *, const int *, const int *);

extern void GMENCODE(const int *, int *, const int *, const int *);

extern void GADECODE(const int *, int *, const int *, const int *);

extern void GAENCODE(const int *, int *, const int *, const int *);

extern void SMDECODE(const int *, int *, const int *, const int *);

extern void SMENCODE(const int *, int *, const int *, const int *);

extern void SADECODE(const int *, int *, const int *, const int *);

extern void SAENCODE(const int *, int *, const int *, const int *);

#endif /* INDEXING_H */
```

5.2 array.c

```
#include "array.h"

int binCoef(const int n, const int m) {
  int result;
  int *work = (int *)calloc((size_t)(m + 1), sizeof(int));
  work[0] = 1;
  for (int i = 1; i <= n; i++) {
    for (int j = (i > m) ? m : i; j > 0; j--) {
      work[j] += work[j - 1];
    }
  }
  result = work[m];
  free(work);
  return (result);
}

void isort(int *cell, const int *n) {
  (void)qsort(cell, (size_t)*n, sizeof(int), icmp);
  return;
}

int icmp(const void *x, const void *y) {
```

```
  const int *ix = (const int *)x;
  const int *iy = (const int *)y;
  return (*ix - *iy);
}

void GVENCODE(const int *location, int *indices, const int *shape,
              const int *rank) {
  *indices = *location + 1;
  return;
}

void GVDECODE(const int *indices, int *location, const int *shape,
              const int *rank) {
  *location = *indices - 1;
  return;
}

void GMENCODE(const int *location, int *indices, const int *shape,
              const int *rank) {
  int nrow = *(shape + 0), icol, irow, iloc = *location;
  irow = (iloc % nrow) + 1;
  icol = (iloc / nrow) + 1;
  *(indices + 0) = irow;
  *(indices + 1) = icol;
  return;
}

void GMDECODE(const int *indices, int *location, const int *shape,
              const int *rank) {
  int nrow = *(shape + 0), irow = *(indices + 0), icol = *(indices + 1),
      iloc = (irow - 1) + (icol - 1) * nrow;
  *location = iloc;
  return;
}

void SMDECODE(const int *indices, int *location, const int *shape,
              const int *rank) {
  int nrow = *(shape + 0), irow = *(indices + 0), icol = *(indices + 1),
      iloc = 0;
  if (irow < icol) {
    iloc = irow;
    irow = icol;
    icol = iloc;
  }
```

```
  int *maxcol = (int *)calloc((size_t)nrow + 1, sizeof(int));
  for (int i = 0; i <= nrow; i++) {
    *(maxcol + i) = i * nrow - i * (i - 1) / 2;
  }
  iloc = *(maxcol + (icol - 1)) + (irow - icol + 1);
  *location = iloc - 1;
  free(maxcol);
  return;
}

void SMENCODE(const int *location, int *indices, const int *shape,
              const int *rank) {
  int nrow = *(shape + 0), irow, icol, iloc = *location + 1;
  int *maxcol = (int *)calloc((size_t)nrow + 1, sizeof(int));
  for (int i = 0; i <= nrow; i++) {
    *(maxcol + i) = i * nrow - i * (i - 1) / 2;
  }
  for (int i = 1; i <= nrow; i++) {
    if (iloc <= *(maxcol + i)) {
      icol = i;
      irow = iloc - *(maxcol + (i - 1)) + (i - 1);
      break;
    }
  }
  *(indices + 0) = irow;
  *(indices + 1) = icol;
  free(maxcol);
  return;
}

void GADECODE(const int *indices, int *location, const int *shape,
              const int *rank) {
  int aux = 1, res = 1, r = *rank;
  for (int i = 0; i < r; i++) {
    res += (*(indices + i) - 1) * aux;
    aux *= *(shape + i);
  }
  *location = res - 1;
  return;
}

void GAENCODE(const int *location, int *indices, const int *shape,
              const int *rank) {
  int r = *rank, iloc = *location + 1, aux = 1, k = 1;
```

```
  for (int j = 1; j < r; j++) {
    aux *= *(shape + j - 1);
  }
  for (int j = r - 1; j > 0; j--) {
    k = (iloc - 1) / aux;
    *(indices + j) = k + 1;
    iloc -= k * aux;
    aux /= *(shape + j - 1);
  }
  *(indices + 0) = iloc;
  return;
}

void SADECODE(const int *indices, int *location, const int *shape,
              const int *rank) {
  int iloc = 1, k = 1, r = *rank;
  int *ind = (int *)calloc((size_t)r, sizeof(int));
  for (int i = 0; i < r; i++) {
    *(ind + i) = *(indices + i);
  }
  (void)isort(ind, rank);
  for (int i = 1; i <= r; i++) {
    k = i + (*(ind + i - 1) - 1) - 1;
    iloc += binCoef(k, i);
  }
  *location = iloc - 1;
  (void)free(ind);
  return;
}

void SAENCODE(const int *location, int *indices, const int *shape,
              const int *rank) {
  int n = *shape, r = *rank, iloc = *location + 1, v = iloc - 1;
  for (int k = r; k >= 1; k--) {
    for (int j = 0; j < n; j++) {
      int sj = binCoef(k + j - 1, k), sk = binCoef(k + j, k);
      if (v < sk) {
        indices[k - 1] = j + 1;
        v -= sj;
        break;
      }
    }
  }
  return;
```

```
}
```

5.3 gvtester.c

```
#include "array.h"

/*
  a vector of shape (10)
*/

int main(void) {
  ARRAY a;
  int length = 10, rank = 1, j = 0;
  int *shape = (int *)calloc((size_t)rank, sizeof(int));
  *shape = 10;
  a.content = (double *)calloc((size_t)length, sizeof(double));
  a.length = &length;
  a.shape = shape;
  a.rank = &rank;
  a.decode = GVDECODE;
  a.encode = GVENCODE;
  printf("========\n");
  printf("GVDECODE\n");
  printf("========\n");
  for (int i = 1; i <= length; i++) {
    (void)a.decode(&i, &j, a.shape, a.rank);
    printf("index %4d location %4d\n", i, j);
  }
  printf("========\n");
  printf("GVENCODE\n");
  printf("========\n");
  for (int i = 0; i < length; i++) {
    (void)a.encode(&i, &j, a.shape, a.rank);
    printf("location %4d index %4d\n", i, j);
  }
  (void)free(shape);
  (void)free(a.content);
  return EXIT_SUCCESS;
}
```

5.4 gmtester.c

```c
#include "array.h"

/*
  a general matrix of shape (4,3)
*/

int main(void) {
  ARRAY a;
  int length = 12, rank = 2, k = 0;
  int *shape = (int *)calloc((size_t)rank, sizeof(int));
  int *indices = (int *)calloc((size_t)rank, sizeof(int));
  *(shape + 0) = 4;
  *(shape + 1) = 3;
  a.content = (double *)calloc((size_t)length, sizeof(double));
  a.length = &length;
  a.shape = shape;
  a.rank = &rank;
  a.decode = GMDECODE;
  a.encode = GMENCODE;
  printf("========\n");
  printf("GMDECODE\n");
  printf("========\n");
  for (int j = 1; j <= *(a.shape + 1); j++) {
    *(indices + 1) = j;
    for (int i = 1; i <= *(a.shape + 0); i++) {
      *(indices + 0) = i;
      (void)a.decode(indices, &k, a.shape, a.rank);
      printf("indices %4d %4d location %4d\n", i, j, k);
    }
  }
  printf("========\n");
  printf("GMENCODE\n");
  printf("========\n");
  for (int i = 0; i < length; i++) {
    (void)a.encode(&i, indices, a.shape, a.rank);
    printf("location %4d indices %4d %4d\n", i, *(indices + 0), *(indices + 1));
  }
  (void)free(shape);
  (void)free(a.content);
  (void)free(indices);
  return EXIT_SUCCESS;
}
```

5.5 smtester.c

```c
#include "array.h"

/*
  a symmetric matrix of shape (4,4)
*/

int main(void) {
  ARRAY a;
  int length = 10, rank = 2, shape = 4, k = 0;
  int *indices = (int *)calloc((size_t)rank, sizeof(int));
  a.content = (CTYPE *)calloc((size_t)length, sizeof(CTYPE));
  a.length = &length;
  a.shape = &shape;
  a.rank = &rank;
  a.decode = SMDECODE;
  a.encode = SMENCODE;
  printf("========\n");
  printf("SMDECODE\n");
  printf("========\n");
  for (int j = 1; j <= shape; j++) {
    *(indices + 1) = j;
    for (int i = 1; i <= shape; i++) {
      *(indices + 0) = i;
      (void)a.decode(indices, &k, a.shape, a.rank);
      printf("indices %4d %4d location %4d\n", i, j, k);
    }
  }
  printf("========\n");
  printf("SMENCODE\n");
  printf("========\n");
  for (int i = 0; i < length; i++) {
    (void)a.encode(&i, indices, a.shape, a.rank);
    printf("location %4d indices %4d %4d\n", i, *(indices + 0), *(indices + 1));
  }
  (void)free(a.content);
  (void)free(indices);
  return EXIT_SUCCESS;
}
```

5.6 gatester.c

```c
#include "array.h"

/*
  a general array of shape (2,4,3)
*/

int main(void) {
  ARRAY a;
  int length = 24, rank = 3, l = 0;
  int *shape = (int *)calloc((size_t)rank, sizeof(int));
  int *indices = (int *)calloc((size_t)rank, sizeof(int));
  *(shape + 0) = 2;
  *(shape + 1) = 4;
  *(shape + 2) = 3;
  a.content = (double *)calloc((size_t)length, sizeof(double));
  a.length = &length;
  a.shape = shape;
  a.rank = &rank;
  a.decode = GADECODE;
  a.encode = GAENCODE;
  printf("========\n");
  printf("GADECODE\n");
  printf("========\n");
  for (int k = 1; k <= *(a.shape + 2); k++) {
    *(indices + 2) = k;
    for (int j = 1; j <= *(a.shape + 1); j++) {
      *(indices + 1) = j;
      for (int i = 1; i <= *(a.shape + 0); i++) {
        *(indices + 0) = i;
        (void)a.decode(indices, &l, a.shape, a.rank);
        printf("indices %4d %4d %4d location %4d\n", i, j, k, l);
      }
    }
  }
  printf("========\n");
  printf("GAENCODE\n");
  printf("========\n");
  for (int i = 0; i < length; i++) {
    (void)a.encode(&i, indices, a.shape, a.rank);
    printf("location %4d indices %4d %4d %4d\n", i, *(indices + 0),
           *(indices + 1), *(indices + 2));
  }
```

```
  (void)free(shape);
  (void)free(a.content);
  (void)free(indices);
}
```

5.7 satester.c

```
#include "array.h"

/*
  a symmetric array of shape (3,3,3)
*/

int main(void) {
  ARRAY a;
  int length = 10, rank = 3, shape = 3, l = 0;
  int *indices = (int *)calloc((size_t)rank, sizeof(int));
  a.content = (double *)calloc((size_t)length, sizeof(double));
  a.length = &length;
  a.shape = &shape;
  a.rank = &rank;
  a.decode = SADECODE;
  a.encode = SAENCODE;
  printf("========\n");
  printf("SADECODE\n");
  printf("========\n");
  for (int k = 1; k <= shape; k++) {
    *(indices + 2) = k;
    for (int j = 1; j <= shape; j++) {
      *(indices + 1) = j;
      for (int i = 1; i <= shape; i++) {
        *(indices + 0) = i;
        if ((i <= j) && (j <= k)) {
          (void)a.decode(indices, &l, a.shape, a.rank);
          printf("indices %4d %4d %4d location %4d\n", i, j, k, l);
        }
      }
    }
  }
  printf("========\n");
  printf("SAENCODE\n");
  printf("========\n");
```

```
  for (int i = 0; i < length; i++) {
    (void)a.encode(&i, indices, a.shape, a.rank);
    printf("location %4d indices %4d %4d %4d\n", i, *(indices + 0),
           *(indices + 1), *(indices + 2));
  }
  (void)free(a.content);
  (void)free(indices);
  return EXIT_SUCCESS;
}
```

References

De Leeuw, J. 2017. "Multidimensional Array Indexing and Storage." 2017. http://deleeuwpdx.net/pubfolders/indexing/indexing.pdf.

Gerber, F., K. Mösinger, and R. Furrer. 2018. "dotCall64: An efficient interface to compiled C/C++ and Fortran code supporting long vectors." *SoftwareX* 7: 217–21.